Working Notes from the 1992 AAAI Workshop on Automating Software Design

Theme: Domain Specific Software Design

July 12-16, 1992
San Jose Convention Center
San Jose, CA

Richard M. Keller (editor)
Sterling Software
Artificial Intelligence Research Branch
MS 269-2
NASA Ames Research Center
Moffett Field, CA 94035-1000

NASA Ames Research Center
Artificial Intelligence Research Branch

Technical Report FIA-92-18

July, 1992

Workshop Notes
AAAI-92 Workshop on Automating Software Design
Theme: Domain-Specific Software Design

July 12-16, 1992
San Jose Convention Center
San Jose, CA

Workshop Committee

Richard M. Keller
Sterling Software
NASA Ames Research Center

Michael R. Lowry
RECOM Technologies
NASA Ames Research Center

David Barstow
Schlumberger Laboratory For
Computer Science

Christopher H. Tong
Rutgers University

Contents

primary

PREFACE

In recent years, there has been an increase in research and development effort aimed at the production of domain-specific software design systems – knowledge-intensive systems that aid in the design of software for specific classes of problems in science, engineering, telecommunications, manufacturing, business, education, and other areas. Despite substantive progress in developing general-purpose software design techniques, the application of these techniques to large, real-world software design tasks has proven difficult. As a result, there is a growing realization that special-purpose, domain-specific techniques will play a critical role in moving research into practice. When restricted to a specific domain, software design systems can avail themselves of additional sources of knowledge and constraints that simplify the overall design process.

The goal of this workshop is to identify different architectural approaches to building domain-specific software design systems and to explore issues unique to domain-specific (vs. general-purpose) software design. Some general issues that cut across the particular software design domain include:

- **Knowledge representation, acquisition, and maintenance:** In building a domain-specific software design system, decisions must be made about what domain knowledge is necessary to support the design task and what formalism to use for the representation. In addition, knowledge-intensive design systems cannot be deployed without seriously addressing the extra burden that comes along with acquiring and maintaining a significant body of domain knowledge.

- **Specialized software design techniques:** By restricting both the domain and the class of software design tasks to be addressed by a system, it becomes possible to utilize specialized design techniques that may simplify and speed up the overall design process.

- **User interaction & user interface:** The typical end-user of a domain-specific software design system is an application specialist, not a programmer with special analytic skills. As a result, domain-specific systems need to pay special attention to the interaction between the user and the system.

We hope that you find the workshop and the papers in this collection both stimulating and informative.

Rick Keller

Richard Keller
Workshop Chair

SCHEDULE

AAI Workshop on Automating Software Design
Theme: Domain Specific Software Design (DSSD)

San Jose, CA Sunday, July 12, 1992

8:30 - 8:45 AM Workshop overview - Richard Keller
8:45 - 9:30 AM Historical perspective and issues - Dave Barstow
9:30 -10:00 AM DARPA DSSA Program Overview - Erik Mettala

10:00-10:30 AM BREAK
10:30-12:15 PM Panel: Approaches to DSSD -- Exploring the generally/power trade-off

Sanjay Bhansali: Generic architectures approach
Neil Goldman: Application generator approach
Neil Iscoe: Domain modeling approach
Discussants: Tong & Keller

12:15- 1:45 PM LUNCH
1:45- 3:30 PM Panel: Practical experience & issues relating to building and fielding DSSD Systems

Elaine Kant: SINAPSE
Richard Keller: SIGMA
Uwe Nonnenmann: KITSS
Discussant: Barstow

3:30- 4:00 PM BREAK
4:00- 5:00 PM Individual DSSD System Presentations
Dorothy Setliff: RT-Syn
Naveen Sharma: PIER
Discussant: Lowry

5:00 PM Closing discussion & wrap-up

Participants

Ashok Agrawala
University of Maryland
Department of Computer Science
College Park, MD 20742
agrawala@cs.umd.edu

Guillermo Arango
Schlumberger Laboratory for Computer Science
8311 North FM 620
Austin, TX 78720, USA

Sidney C. Bailin
CTA Incorporated
6116 Executive Boulevard Suite 800
Rockville, MD 20852
sbailin@cta.com

Prof. Paul D. Bailor
Air Force Inst. of Technology
AFIT/ENG
WPAFB, OH 45433-6583
pbailor@galaxy.afit.af.mil

David Barstow
Schlumberger Laboratory for
Computer Science
50, Avenue Jean Jaures,
B.P. 620-05
92542 Montrouge Cedex
FRANCE
barstow@slcse.slb.com

Sanjay Bhansali
Knowledge Systems Laboratory
Dept. of CS
701 Welch Road, Bldg. C
Stanford University
Stanford, CA 94304
bhansali@sumex-aim.stanford.edu

Christine Braun
GTE Federal Systems
15000 Conference Center Dr.
Chantilly, VA 22021

Steve Chien
JPL M/S 525-3660
4800 Oak Grove Drive
Pasadena, CA 91109-8099
chien@ai-cyclops.jpl.nasa.gov

Douglas Dankel
E301 CSE, CIS
Univeristy of Florida
Gainesville, FL 32611
ddd@cis.ufl.edu

Premkumar Devanbu
2B417, AT&T Bell Labs
600 Mountain Ave.
Murray Hill, NJ 07974
prem@research.att.com

Richard D'Ippolito
Software Eng. Institute
Carnegie Mellon University
Pittsburg, PA 15213-3890
rsd@sei.cmu.edu

Thomas Ellman
Dept. of Computer Science
Hill Center for Mathematical Science
Rutgers University
New Brunswick, NJ 08903
ellman@cs.rutgers.edu

Henrik Eriksson
Section on Medical Informatics
Stanford University MC
Stanford, CA 94305-5479
eriksson@sumex-aim.stanford.edu

Lee D. Erman
Cimflex Teknowledge
1810 Embarcadero Road
P.O. Box 10119
Palo Alto, CA 94303
lerman@teknowledge.com

Leona Fass
P.O. Box 2914
Carmel, CA 93921

Mark Feblowitz
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254
feblowitz@gte.com

Brian Gaines
The University of Calgary
Dept. of Computer Science
2500 University Drive, NW
Calgary, Alberta CANADA T2N 1N4
gaines@cpsc.ucalgary.ca

R. Gastner
Forwiss
Am Weichselgarten 7,
D-8520 Erlangen
West Germany
gastner@forwiss.uni-erlangen.de

Neil Goldman University of Southern CA
Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6995
goldman@isi.edu

Cordell Green
Kestrel Institute
3260 Hillview
Palo Alto, CA 94304
green@kestrel.edu

Sol Greenspan
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254
greenspan@gte.com

Robert Hall
AT&T Bell Laboratories
600 Mountain Aven, Rm 3D-458
P.O. Box 636
Murray Hill, NJ 07974-0636
hall@allegra.att.com

Frederick Hayes-Roth
Cimflex Teknowledge
1810 Embarcadero Road
P.O. Box 10119
Palo Alto, CA 94303
rhayes-r@teknowledge.com

Neil Iscoe
EDS Research - Austin Lab.
1601 Rio Grande Ste. 500
Austin, TX 78701
iscoe@austin.eds.com

Hermann Kaindl
Siemens AG Osterreich
Program & System Engineering
Geusaugasse 17
Vienna, Austria 1030, Europe
kaih@siegnd.siemens.co.at

Elaine Kant
Schlumberger Lab. for
Computer Science
P.O. Box 200015
Austin, TX 78720
kant@slcs.slb.com

Richard Keller
AI Research Branch
M/S 269-2
NASA Ames Research Center
Moffett Field, CA 94035
keller@ptolemy.arc.nasa.gov

Kirby Keller
McDonnell Aircraft Company
Dept. 313, MC 106-5205
P.O. Box 516
St. Louis, MO 63166-0516
keller@aicenter.mdc.com

Kenneth Lee
Software Eng. Institute
Carnegie Mellon University
Pittsburg, PA 15213-3890
kl@sei.cmu.edu

Vladimir Lovitsky
Software Engineering Dept.
Institute of Radioelectronics
Kharkov, Ukraine

Mike Lowry
AI Research Branch
NASA Ames Research Center
M/S 269-2
Moffett Field, CA 94035-1000
Lowry@ptolemy.arc.nasa.gov

Neil Maiden
Department of Business College
City University
London ECIV OHB, UK
cc559@city.ac.uk

David McAllester
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge Mass. 02139

LTC Erik G. Mettala, Ph.D
Deputy Director
Software & Intelligent Systems
Technology Office
Defense advanced Research
Projects Agency
3701 N. Fairfax Drive
Arlington, Va 22203 1714
(703) 696-2219
mettala@darpa.mil

Penny Nii
Knowledge Systems Laboratory
Dept. of CS
701 Welch Road, Bldg. C
Stanford University
Stanford, CA 94304
nii@sumex-aim.stanford.edu

Uwe Nonnenmann
AT&T Bell Labs
600 Mountain Avenue
Murray Hill, NJ 07974
un@research.att.com

W. F. Punch
Michigan State University
Depart. of Computer Science
A-714 Wells Hall
East Lansing, MI 48824-1027
punch@cps.msu.edu

John S. Robinson
Air Force Inst. of Technology
AFIT/ENG
WPAFB, OH 45433-6583

M. Schmalz
Dept. of Computer and Info. Sci.
Univ. of Florida
Gainesville, FL 32611
msz@mosquito.cis.ufl.edu

Dorothy Setliff
Electrical Engineering Dept.
University of Pittsburgh
Pittsburgh, PA 15261
dottie@jaguar.ee.pitt.edu

Naveen Sharma
ICM
Dept. of Math/CS
Kent State University
Kent, OH 44240
sharma@mcs.kent.edu

Mildred Shaw
The University of Calgary
Dept. of Computer Science
2500 University Drive, NW
Calgary, Alberta CANADA T2N 1N4
shaw@cpsc.ucalgary.ca

Nancy Solderitsch
Paramax Systems Corporation
Unisys
70 E. Swedesford Road
Paoli, PA 19301
nancy@prc.unisys.com

Kevin Stanley
McDonnell Aircraft Company
Dept. 313, MC 106-5205
P.O. Box 516
St. Louis, MO 63166-0516
stanley@aicenter.mdc.com

Chris Tong
Dept. of Computer Science
Hill Center
Busch Campus
Rutgers University
New Brunswick, NJ 08903
ctong@cs.rutgers.edu

Mike Uschold
AI Applications Inst.
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
mfu@aiai.edinburgh.ac.uk

Steve Vestal
Honeywell
S&RC MN65-2100
3660 Technology Drive,
MN 55418
vestal@src.honDr.

Vladimir Lovitsky
c/o Pat Pearce
Department of Computing
Polytechnic South West
Drake Circus
Plymouth
Devon
PL4 8AA
United Kingdomeywell.com

Kerstin Voigt
Computer Science Dept.
Rutgers University
New Brunswick, NJ 08903
voigt@cs.rutgers.edu

Wayne Walker
E301 CSE, CIS
University of Florida
Gainesville, FL 32611
ww0@cis.ufl.edu

Kurt Wallnau
Paramax Systems Corporation
Unisys
70 E. Swedesford Road
Paoli, PA 19301
wallnau@Cards.COM

Richard Waters
Mitsubishi Electric Research Laboratories
201 Broadway
Cambridge MA 02139
dick@merl.com

Developing Satellite Ground Control Software through Graphical Models

Sidney Bailin, Scott Henderson, and Frank Paterra

CTA Incorporated
Rockville, MD

Walt Truszkowski

NASA/Goddard Space Flight Center
Greenbelt, MD

1 Introduction

The old maxim goes, "A picture is worth a thousand words"—ten thousand, if you believe Larkin and Simon (1987). Most people, when faced with the problem of understanding the behavior of a complicated system, resort to the use of some picture as an aid in thinking about the system. Barwise and Etchmendy (1991) make a strong case for the effectiveness of diagrams, over and above other representations, in certain problem solving situations.

This paper discusses a program of investigation into software development as graphical modeling. The goal of this work is a more efficient development and maintenance process for the ground-based software that controls unmanned scientific satellites launched by NASA. The main hypothesis of the program is that modeling of the spacecraft and its subsystems, and reasoning about such models, can—and should—form the key activities of software development; and that by using such models as inputs, the generation of code to perform various functions (such as simulation and diagnostics of spacecraft components) can be automated. Moreover, we contend that automation can provide significant support for reasoning about the software system at the diagram level.

The outline of this paper is as follows. We describe the application domain in the next section, and the graphical modeling technique in Section 3. Sections 4 and 5 describe the approach to generating diagnostic and simulator software from these models. In Section 6 we describe the work we are doing in automated reasoning about the diagrams. Finally, in Section 7, we summarize what we think are the prospects for this program, the key issues, and major risks and unknowns.

2 The Domain: the Intelligent Ground System

Simulation and diagnostics play a key role in a satellite control center. They support the two principal activities of the control center—commanding and monitoring the spacecraft. Development of command loads prior to spacecraft launch employs simulation to verify their proper operation. Monitoring involves fault detection, isolation, and recovery when telemetry values received from the spacecraft fall outside of defined limits. In our work implementing a testbed for an advanced control center, which we call the Intelligent Ground System (IGS), we found that simulation and diagnosis activities tend to derive from the same set of knowledge, namely models of the spacecraft components. An integrated approach, in which diagnosis and simulation are both driven by the same run-time models, seems feasible to us; at this point, however, we are aiming at a less ambitious goal, which is the generation of distinct programs to support the respective functions from the same graphical model. We can view this as "design time integration" rather than "run time integration."

The importance of such models is a result of an object-oriented system architecture, which is one of the defining characteristics of the IGS. The object-oriented architecture describes the IGS as a model of its environment. This environment consists primarily of the spacecraft, its subsystems, and payload, and the users of the IGS (the Flight Operations Team, or FOT), who are divided into several distinct roles within the control center. The environment may also be viewed as including the communications systems through which the IGS and the satellite interact, and various other ground systems to which a control center is typically connected. Each of these environmental elements is represented as a distinct object in the IGS. This approach enables us to make the IGS "intelligent" by making each such object a knowledge-based system in its own right, with its own simulation capability, diagnostic capability, etc.

There are various consequences of this architecture for the operation of the IGS, including the need for a cooperative framework, and for an intelligent user interface. The object-oriented architecture defines the IGS as a collection of interacting knowledge-based systems. This interaction models the interaction

found in the system's environment, but it must also include means for cooperative problem solving among the system's components. For example, the successful diagnosis of telemetry anomalies may require interaction between the diagnostics of several subsystems. Thus, a key requirement of the IGS is a set of problem-solving protocols through which multiple knowledge-based systems, in conjunction with the FOT, can converge towards a goal. A framework for such cooperation is described by Bailin *et al* (1989).

The need for an intelligent user interface follows from the fact that he IGS does not operate autonomously—the health and safety of the spacecraft precludes such an approach. The FOT are active players in the cooperative process just described. Thus, the IGS must model the FOT roles in a way that a) facilitates communication between the human and the machine, and b) enables the system to interpret human actions within the cooperative protocols.

2.1 Implications for Software Development

The IGS architecture makes everything a model. The software models the states, behaviors, and interactions of elements in its environment. Given this role for the software, it seems appropriate to look for a language in which such information can be made explicit. Graphical modeling of objects, their behaviors, and their interactions is an obvious choice for such a language; there is nothing new in our advocacy of diagrams to express such information. Our contention, which may be more questionable, is that the real complexity of the software lies in the interactions expressed by the graphical models, not in the implementation details of the eventual code.

We contend that the structure of the implemented code, for at least certain functions of the IGS—specifically, simulation and diagnosis—is sufficiently well understood to permit us to generate it automatically, and therefore to allow us to redefine the development process as one of developing and reasoning about the graphical models. The following sections describe the progress we have made to date in demonstrating this idea. Similar ideas have been put forward in a recent article by Harel (1992).

The more advanced IGS functions—the cooperative framework and the intelligent user interface—go beyond our current view of what can be automatically generated from graphical models. The reason is simple: we do not yet have an adequate understanding of these functions. Our work on the IGS is attempting to make inroads into these areas, especially the cooperative framework, but this work is still exploratory. We expect that with the definition of cooperative protocols, code implementing such protocols would be provided as reusable library assets. It is conceivable, therefore, that they could form a part of the automated development framework towards which we are working.

3 The Graphical Models

The diagrams consist of objects described by behavioral annotations and connected to each other by influence paths. Each object has a set of state variables, some of which serve as input ports (receiving influences), some of which serve as output ports (creating influences), and some of which are internal to the object. In translating such a diagram into a simulator, the influence paths are implemented as data flows. The influences paths may, however, correspond to the transfer of physical attributes, e.g., heat transfer, in the modeled system itself. Thus, the graphical representation is somewhat different from the conventional notion of a dataflow diagram.

The object behaviors are described in terms of states and transitions, but the representation is more powerful than that of a finite state machine. Each internal and output state variable has a finite number of "transitions" associated with it, but each such transition is a mathematically specified function. Thus, the domain of each transition is a set of possible initial state values; the resulting state, and any corresponding outputs, are a function of the initial state. This function may be defined in a piecewise fashion: that is, the set of possible initial states may be partitioned into a finite number of subsets, and the transition may then be defined on each subset by an appropriate expression. This seems to be similar to the approach recently advocated by Parnas (1990), in which tables are used to specify the discontinuities often present in functions that software is required to compute.

Components are stored in a library, so that they may be reused in many applications. Components are typed, and intuitively fall into a class hierarchy, although the library system does not yet support inheritance. Components may contain subcomponents as well as the state variables discussed above. In such cases, the interconnection of the subcomponents via influences forms part of the parent component description. There are no "systems" *per se* in the library: everything is a component. A system can be stored in the library as a new component, in which case it is available for use as a component in a still larger system in the future.

4 Generating Diagnostic Rules: the Knowledge from Pictures System

The Knowledge from Pictures (KFP) tool builds a knowledge base to perform fault detection, isolation, and recovery from a diagram of the monitored system.

The generated knowledge base takes the the form of facts and rules in the C Language Integrated Production System (CLIPS), an expert system shell developed by NASA/Johnson Space Center. The diagram is also used as the basis for the user interface of the diagnostic system.

Assertions derived from the behavioral descriptions of the diagram's components are used to determine when a component is in a state other than those in its definition (for example, a temperature sensitive object operating outside of its design temperature range). When such a situation has been detected, a fault has occurred. Alarms are defined as collections of component states. In the generated knowledge base, each alarm condition is represented by a CLIPS rule.

The rules generated by KFP use the influence paths shown in the diagram to isolate failed components. When an alarm is detected, a search begins for the faulted object causing the alarm. The search is performed by tracing back through the paths of influence that are input to the alarming object. The influence paths form a collection of chains of objects that either directly or indirectly influence the components contributing to the alarm. The tracing is performed via a collection of rules that examine the objects in each path. When these rules fire, they use information about the known states of the object being examined, and the states of the objects that influence it, to determine whether the examined object is behaving correctly. If the object being examined is not in the correct state, then the fault has been isolated. If it *is* in the correct state, the objects that influence it are examined next.

After a fault has been detected and isolated, the recovery phase begins. At present the recovery phase is represented by a template for recovery rules—one for each fault/object pair. The action part these rules must be filled in by the knowledge engineer.

In KFP, the diagram of the system being monitored is also intended to serve as the basis for the diagnostic system's user interface. The control center operator should see a display of the system as a graphical model, with the status of its components expressed through color coding or similar conventions. The current KFP tool does not do this, but the concept has been demonstrated by another prototype system, the Generic Spacecraft Analyst Assistant (GenSAA).[1] Our plan is to integrate KFP with the next version of GenSAA by the end of this year.

[1] The GenSAA project is directed by Peter Hughes of NASA/Goddard's Automation Technology Section (Code 522.3).

5 Generating Simulator Software: the Multi-Aspect Simulation Tool

Our generic simulation architecture is based on the connection manager approach described in the Software Engineering Institute's (SEI) recommendations for flight simulators (Lee, 1990). In this approach, the influences between objects are simulated as data flows, and the data flows are implemented by connection managers—objects whose specific role is to manage the connections between application objects. The benefit of this approach is that the application objects themselves remain ignorant of the context in which they are used, and thus can be reused in quite different contexts.

In the Multi-Aspect Simulation Tool (MAST) we have extended SEI approach by independently formalizing each aspect of a component's behavior, by integrating work on discrete event simulation done by Zeigler (1990), and by implementing the design using the object-oriented techniques of multiple inheritance and virtual base classes.

Simulations typically represent system behavior along several dimensions. In MAST these dimensions are rendered by the interactions of independent aspect managers. Each manager is concerned with different component attributes. A gravitational manager, for example, is concerned with a component's position and mass, but not with its shape or color. All components subject to a manager appear to that manager with the same form, regardless of their actual structure. The manager can therefore assess and manipulate the components in terms of this standard form, oblivious to interactions occurring within the component with other aspects of its behavior. For example the gravitational manager should be able to change a component's position oblivious to the fact that the change also modified the component's shape. This homomorphy is available in C++ through multiple inheritance and virtual methods.

MAST integrates both discrete event simulation and continuous simulation techniques. For continuous aspects of the simulation, the associated aspect manager schedules re-evaluations at regular intervals of simulated time. These intervals can be decreased during the simulation to enhance the fidelity of the behavior rendered for a particular passage, and then lengthened to speed the simulation through a passage where little is changing. For discrete aspects of the simulation, the associated aspect manager schedules re-evaluation at the time of the most imminent event known. When that simulated time is achieved, the aspect manager executes the associated event, propagates its effects, and then computes the next imminent event for scheduling. A central simulation manager decides how to advance the logical clock by perusing each manager's

schedule. The clock is advanced to the most imminent re-evaluation time, and the managers who are scheduled for that time are executed.

Although not yet implemented, we view it as a straightforward task to generate the connection management code automatically from the graphical models, and plan to do so in the near future. Generating the specific algorithms of each aspect manager, using the associated behavior specifications from the graphical model, would be a far more difficult task, which we do not plan to tackle in the near future.

6 Reasoning about the Diagrams

We have been working for several years on an automated reasoning system that takes diagrams as input. The GROVER system attempts to interpret the diagram as a high-level description of a proof plan, and it attempts to carry out the plan using an underlying "conventional" theorem prover (Barker-Plummer and Bailin, 1992). Recently we have begun to apply these ideas to the problem of reasoning about software. The graphical models that we discussed in the previous sections are interpreted by this (as yet unnamed) tool as plans for proving assertions about the software design.

The particular type of assertions processed by this tool grew out of an actual experience in debugging part of the IGS testbed. In testing a particular simulator program it was found that the behavior of the system was not as expected, but no errors could be found in any of the simulator components. The problem turned out to be one of missing connections between objects in the simulator. Since the simulator architecture keeps each object autonomous—completely ignorant of the objects to which it is connected in a given application—the absence of these connections did not result in any anomalous behavior on the part of any object, but the system itself was not behaving as expected.

Thus we decided to apply the planning concept to verifying statements of the form, "If event x occurs at object A then event y will occur at object B." The planner takes event y at B as a goal, and tries to construct a plan that starts from event x at A as an initial condition (typically, various other context conditions are specified as well). A goal is reduced to subgoals by traversing the connections specified in the diagram: if a goal state in an object D follows, according to D's behavior description and the connections specified in the diagram, from a certain state in object C, then this state in object C becomes a subgoal of the goal state. A failed plan, when presented to the developer, serves to identify missing connections that may have been overlooked in defining the system.

We have noticed a similarity in the logic of this planner and that of the KFP tool, which similarly traces back through the influence paths in the diagram in generating fault isolation rules. We have not studied this similarity in enough detail to decide whether the two tools can make use of a single "influence traverser" mechanism, but there seems to be some promise of this.

7 Conclusions

We have made a start at what we hope will become an integrated graphical modeling and development system, in which software development becomes synonymous with defining and reasoning about graphical models. The prospects for such an integrated environment are based on a few empirically perceived similarities:

- Similarity between the information used to simulate a system and that used to diagnose faults
- Similarity between the logic used to reason about system behavior during development, and that used to diagnose faults during operation (backward chaining over influence paths)
- Similarity in the program structure of specific simulators and specific diagnostic systems, which has allowed us to define generic architectures for each of these applications

We noted in Section 2 that the full IGS concept includes a lot more than a collection of simulation and diagnostic programs. We are not yet in a position to say whether these advanced capabilities can be accommodated in our application development framework. Even if they are not, however, the current framework raises the level of abstraction at which a significant amount of development for a control center is performed.

Within the scope of the current framework, there are perhaps two major open issues: 1) the impact of scale-up on the performance of the generated code, and 2) the feasibility of automated reasoning about additional aspects of the models.

The efficiency of the generated fault detection, isolation, and recovery rules for a large, complex system is an open issue. The examples we have worked with to date in KFP have been obtained from actual systems (either existing or being developed), but they are very small subsets of these systems. There is a solid basis of real-time scheduling theory (e.g., rate-monotonic scheduling) with which we can address scale-up performance issues for the generated simulator code, but we lack such a firm basis for a rule-based diagnostic system. The solution to this problem may be to evolve to a more thoroughly model-based approach to diagnosis, in which there is no production rule interpreter at all. This would, in

addition, permit a greater degree of integration between the diagnostic and the simulator code.

An open issue concerning reasoning about the models is whether automation can support reasoning about issues other than the pre-condition/post-condition behaviors currently addressed. One major area that we would like to investigate is support for reducing the state space of a set of interacting components. This problem arises in "reachability analysis," in which one tries to prove (or at least to convince oneself) that no unexpected states are entered. In the area of communications protocols, this has proven to be a difficult but necessary process that can be supported by a variety of heuristic techniques, some of which are automated (Holzman, 1992; Lin and Liu, 1992)

References

Bailin, S., Moore, J., Hilberg, R., Murphy, E., and Baher, S., 1989. A logical model of cooperating rule-based systems. *Telematics and Informatics*, Vol. 6 Nos. 3/4, pp. 331-349.

Barker-Plummer, D. and Bailin, S. Proofs and pictures: proving the diamond lemma with the GROVER theorem proving system. *AAAI Symposium on Reasoning with Diagrammatic Representations*, March 1992.

Barwise, J. and Etchmendy, J., 1991. Visual information and valid reasoning. Preprint.

Harel, D., 1992. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, January 1992.

Holzman, G., 1992. Protocol design: redefining the state of the art. *IEEE Software*, January 1992.

Larkin, S and Simon, H., 1987. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11, pp 65-100.

Lee, K. *et. al.*, 1990. An OOD paradigm for flight simulators, 2nd edition. Technical Report of the Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

Lin, F. and Liu M., 1992. Protocol validation for large-scale applications. *IEEE Software*, January 1992.

Parnas, D., Asmis, G., and Madey, J., 1990. Assessment of safety-critical software. Technical Report 90-295, ISSN 0836-0227. Telecommunications Research Institute of Ontario. Queens University, Kingston, Ontario.

Zeigler, B., 1990. Object-oriented simulation with hierarchical, modular models. New York: Academic Press.

Formalization and Visualization of Domain-Specific Software Architectures

Paul D. Bailor, David R. Luginbuhl, and John S. Robinson

Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45433
pbailor@galaxy.afit.af.mil
(513) 255-3708

1 INTRODUCTION

This paper describes a domain-specific software design system based on the concepts of software architectures engineering [Lee and others, 1991] and domain-specific models and languages [Prieto-Díaz and Arango, 1991]. In this system, software architectures are used as high level abstractions to formulate a domain-specific software design. The software architecture serves as a framework for composing architectural fragments (e.g., domain objects, system components, and hardware interfaces) that make up the knowledge (or model) base for solving a problem in a particular application area [Lee and others, 1991]. A corresponding software design is generated by analyzing and describing a system in the context of the software architecture [Lee and others, 1991]. While the software architecture serves as the framework for the design, this concept is insufficient by itself for supplying the additional details required for a specific design. Additional domain knowledge is still needed to instantiate components of the architecture and develop optimized algorithms for the problem domain. One possible way to obtain the additional details is through the use of domain-specific languages. Thus, the general concept of a software architecture and the specific design details provided by domain-specific languages are combined to create what can be termed a domain-specific software architecture (DSSA).

2 DESCRIPTION OF DOMAIN SPECIFIC SOFTWARE DESIGN SYSTEM

The overall goal of our research is to prototype the technology required to formally specify, design, and develop an Ada application system using the DSSA approach described above. A key part of this effort is the creation and use of formal, domain-specific languages to generate software architectures whose architectural fragments and associated composition rules are maintained in a formal knowledge base of objects. These languages allow definition of the objects making up the components of the DSSA in terms of the components' structural and behavioral properties. Additionally, the domain-specific languages are used to compose the defined objects into a corresponding software design and resulting Ada implementation. In fact, the production rules of the grammar for the domain-specific language can serve as the basis for system composition. This is consistent with the approach suggested by Batory for composing hierarchical software systems with reusable components [Batory and O'Malley, 1992].

From a user interface perspective, software engineers use the domain-specific languages to define object classes and object composition rules to be placed into the knowledge base of objects. Alternately, application specialists (system end-users) use the languages to introduce new object instances and to compose object instances that currently exist in the knowledge base. Within the knowledge base, the objects and composition rules are maintained as executable, formal specifications providing the software engineer and application specialist with the ability to rapidly prototype and validate desired system behaviors without having to build Ada components first.

In addition to the domain-specific languages, some type of object base language is required to formalize the architecture and corresponding design representation. Such an object base language would also provide the ability to analyze and manipulate the objects defined and composed by the domain-specific language. This object base definition and manipulation language is used for the continued development of the domain-specific design and corresponding Ada software components. That is, the manipulation language is used to

formally manipulate the architectural objects to obtain a corresponding Ada design representation and Ada implementation of that object. Also, the object base manipulation language is the key to developing an application system that is composed of existing and validated Ada software components.

Thus, a formal framework for defining software architectures and domain specific languages would have to consist of the components listed below. The relationship or general configuration of the components is shown in Figure 1.

1. A *formalized object base* that serves two functions:
   (a) Formal specification of the concept of a software architecture consisting of a set of general abstractions associated with software architectures and a mathematical model of these abstractions.
   (b) Formal specifications of architectural fragments and instances of these fragments developed through the use of domain-specific languages as well as the object base manipulation language.
2. Formal specifications of *domain-specific languages* for describing and manipulating objects in the DSSA.
3. An *Ada development capability* that uses the formal specification of the architectural components to generate Ada components and allows for the composition of existing and validated Ada components into an application system.
4. A sophisticated *user interface* for both the application specialist and the software engineer. Note that visualization capabilities for both the domain-specific language constructs and the object base are highly desirable components of the user interface.

For this research effort, a prototype implementation of the technology will be done using the Software Refinery^{TM} Environment [Systems, 1990] that consists of the following components.

1. The Refine wide-spectrum language.
2. The Refine formal object base that is analyzed and manipulated via the Refine language.
3. The Dialect tool that allows for the definition of formal languages whose syntactical structures are directly mapped to objects in the object base. Note that this mapping is done in such a way that an abstract syntax tree relationship is maintained between the language components and corresponding objects in the object base. This relationship provides a significant advantage for language transformation purposes.
4. The Intervista tool that provides an X-windows based capability for graphical interaction with the object base.

Figure 2 graphically depicts the Refine framework. It provides an ideal platform to prototype the proposed DSSA technology. The Dialect and Intervista Tools are used for the User Interface aspect as they provide the means to define domain-specific languages, map domain language structures to a formal object base, and visualize both the domain language structures and the software architecture. The Refine language provides the means to manipulate objects in the object base for performing operations such as transforming the formalized objects into Ada components, analyzing the object's behavior for validation/verification purposes, and composing sets of objects into a higher level application system. An important advantage of the Refine framework is that it reduces tool development time to zero. Thus, it allows the research to focus on the development of the new technology immediately.

3 Research Issues

The domain-specific software design system described above has several important research issues associated with it that we are currently attempting to address.

1. What are the abstractions associated with the concept of a software architecture, and how can we formally model these abstractions?
2. What is the feasibility of developing the required domain-specific languages? There are a number of relevant Air Force application domains that have already been analyzed and at least partially structured using the concepts of a software architecture; for example, the electronic combat domain of The Joint Modeling and Simulation System (J-MASS) [ASD/RWWW, 1990, ASD/RWWW, 1991], the C^{3}I domain [Plinta and Lee, 1989], and the radar tracking domain [Jensen and Ogata, 1991]. Additionally, the DARPA Domain-Specific Software Architecture project is funding research in an attempt to define software architectures in four application areas. All of these could serve as candidates for development of domain-specific languages; however, we must first address three important sub-issues:
   (a) How difficult is it in general to encode the domain-specific knowledge required to compose objects in the domain into the production rules of a grammar? Alternatively, how difficult is it to develop and formalize the domain-specific knowledge required for this and place it into a knowledge base of composition rules?

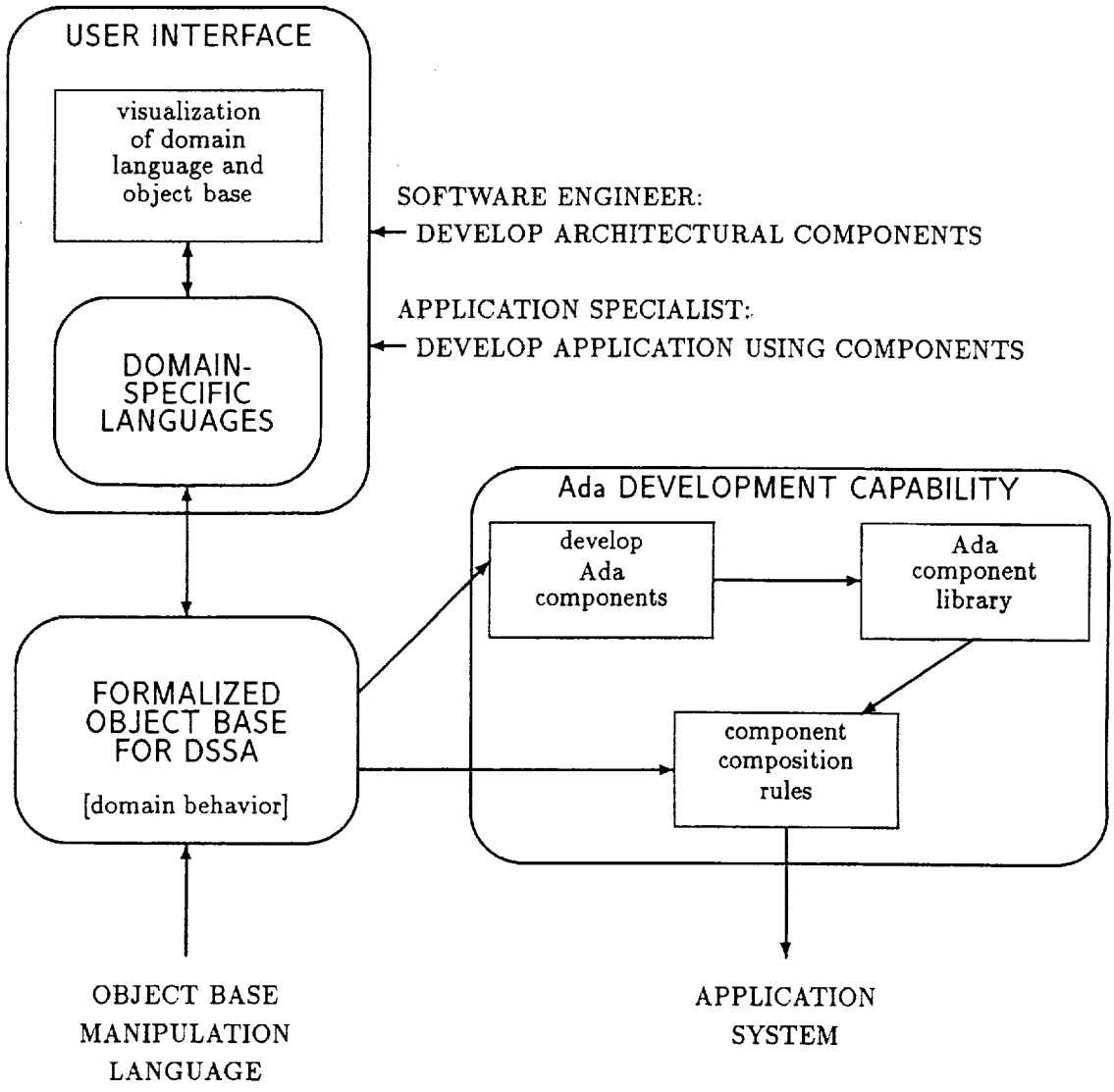


Figure 1: General Configuration for Formalizing a DSSA

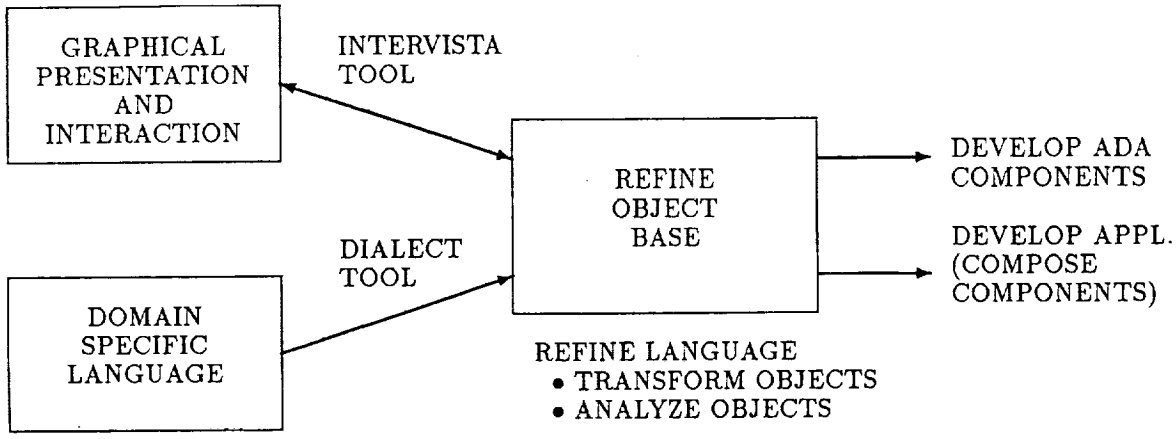


Figure 2: Refine System Framework

(b) Can the same domain-specific language be used by both the software engineer and the application specialist?

(c) Is there a core of language constructs that are common to all domain languages?

3. What is the feasibility of using a knowledge-based transformation system to develop a highly visual interface to the domain-specific software design system that is useful to both the software engineer and the application specialist?

We have focused our short-term research objectives towards addressing the above issues first. Specifically, the research objectives for the first two years of this effort are to:

1. Define the abstractions associated with the concept of software architectures and develop a mathematical model of these abstractions. The Refine system will be used to prototype and analyze formal models of software architectures.

2. Analyze a part of the electronic combat domain and develop the required formal domain language(s).

3. Use the Software Refinery Environment to build a working prototype of our DSSA system that includes the ability to build a formal object base of architecture fragments and to apply composition rules to the fragments to construct domain-specific software designs in the form of a DSSA.

4. Develop visualizations of both the formal domain language(s) and the object base.

In the following years, we expect the major concentration to be on using the formalized object base as a basis for developing the corresponding Ada components and as a basis for composing an application system within a domain. Additionally, methods and tools for scaling the technology up for large scale applications will be investigated.

4 SUMMARY

We feel the proposed research can have a significant impact on the methodologies for implementing the concepts of software architectures and domain-specific software design, especially in the areas of formalizing software architectures and formalizing the application of domain-specific languages to software specification and design. Additionally, this research should provide much insight into the process of object-oriented development of validated and reusable Ada components that can be quickly and validly composed into an application system for a particular domain.

References

[ASD/RWWW, 1990] ASD/RWWW. Joint Modeling and Simulation System (J-MASS): System Concept Document. Technical report, CROSSBOW-S Architecture Technical Working Group, December 1990.

[ASD/RWWW, 1991] ASD/RWWW. Software Structural Model Design Methodology. Technical report, Architecture Technical Working Group, June 1991.

[Batory and O'Malley, 1992] Don Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems With Reusable Components. Technical Report TR-91-22, Department of Computer Sciences, University of Texas at Austin, Austin, Texas, January 1992.

[Jensen and Ogata, 1991] Paul S. Jensen and Lori Ogata. Final Report for Automatic Programming Technologies for Avionics Software (APTAS). Technical Report LMSC-P000001, Lockheed Software Technology Center, Palo Alto, California, July 1991.

[Lee and others, 1991] Kenneth J. Lee et al. Model-Based Software Development (Draft). Special Report CMU/SEI-92-SR-00, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 30 1991.

[Plinta and Lee, 1989] Charles Plinta and Kenneth Lee. A Model Solution for the C^3I Domain. In *Tri-Ada Conference*, pages 56–67. New York: ACM Press, 1989.

[Prieto-Díaz and Arango, 1991] Prieto-Díaz and Arango. *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press : California, 1991.

[Systems, 1990] Reasoning Systems. Refine User's Guide. Reasoning Systems, Inc., 1990.

THE KASE APPROACH TO DOMAIN-SPECIFIC SOFTWARE SYSTEMS

Sanjay Bhansali and H. Penny Nii

Knowledge Systems Laboratory
Stanford University
701 Welch Road, Bldg. C, Palo Alto, CA 94304
bhansali@sumex-aim.stanford.edu
nii@sumex-aim.stanford.edu

1. Introduction

Designing software systems, like all design activities, is a knowledge-intensive task. Several studies, (e.g. [Adelson & Soloway, 1985; Guindon, Krasner, & Curtis, 1987]) have found that the predominant cause of failures among system designers is lack of knowledge – knowledge about the application domain, knowledge about design schemas, knowledge about design processes. The goal of domain-specific software design systems is to explicitly represent knowledge relevant to a class of applications and use it to partially or completely automate various aspects of the design activity for designing systems within that domain. The hope is that this would reduce the intellectual burden on the human designers and lead to more efficient software development.

In this paper, we present a domain-specific system built on top of KASE, a knowledge-assisted software engineering environment being developed at the Stanford Knowledge Systems Laboratory. We introduce the main ideas underlying the construction of domain specific systems within KASE, illustrate the application of the idea in the synthesis of a system for tracking aircrafts from radar signals, and discuss some of the issues in constructing domain-specific systems.

2. Domain Specific Software Systems using KASE

KASE is a knowledge-based software development environment that is designed to provide *active* assistance in the design of software systems. Some of the basic characteristics of the KASE environment are: a domain-independent representation mechanism for software architectures, a graphical interface that permits smooth navigation between different views of a software system [Guindon, 1992], an integrated editor that permits modifications to the architecture from any view, and a constraint checker that can help a user maintain various syntactic and stylistic constraints between different components of the architecture[Nii, Aiello, Bhansali, Guindon, & Peyton, 1991].

The construction of domain specific software systems in KASE involves the identification of a *generic problem* or task, a *generic architecture* suitable for the task, a model of the application domain in terms of primitive entities (e.g. object, relations, events), and a set of *customization tools* that can be used to construct a specific system for a particular problem.

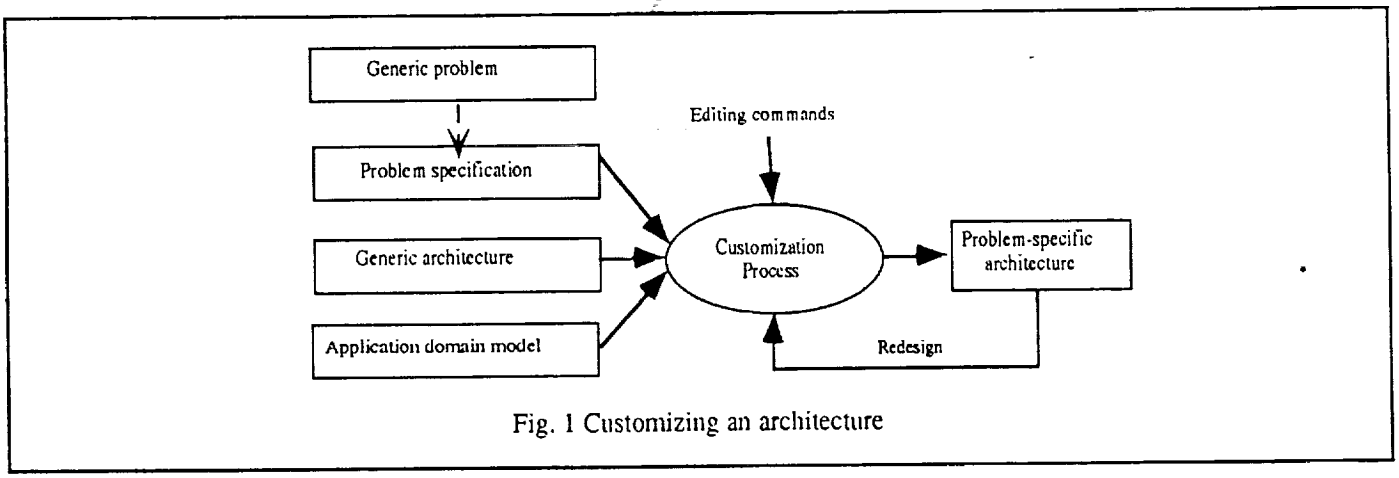


Fig. 1 Customizing an architecture

As shown in figure 1, the software design activity consists of instantiating the generic architecture with respect to a given problem statement and the domain model using the customization tools and results in the creation of a problem-specific architecture. We call this process *customization* - customize a generic architecture to fit an application.

A generic problem represents a class of problems. By identifying problem classes, one can design knowledge representation schemes, architectures, and reasoning processes which are appropriate for the general problem, and reuse them for several different problem instances. The specification of a generic problem results in the creation of a problem schema which specifies the high-level structure of a problem specification. A schema has certain *roles* which represent the parameters of the problem, and *constraints* on the values of the roles. Instantiating these roles with specific values results in the creation of a specific problem specification.

Figure 2 shows the schema for an example generic problem: tracking a set of mobile objects by interpreting signals that are being continually generated by the objects. (This generic problem can be instantiated, e.g. to the problem of tracking aircrafts from radar and voice signals (Brown, Schoen, & Delagi, 1986) or tracking ships from sonar data (Nii, Feigenbaum, Anton, & Rockmore, 1982)). This problem has three parameters: (i) the specification of the input signal(s); (ii) the main body or functional description of the problem in the form of an extremely high-level program; and (iii) certain characteristics of the domain and the environment. The constraints on the schema roles are specified by specifying a grammar for instantiating the roles.

Associated with each generic problem is a set of (possibly one) generic architectures, which can be used to create a system for solving instances of the generic problem. A generic architecture is a collection of *parameterized modules* and intermodular dependencies. A parameterized module is a logical collection of software entities like procedures, types, etc. in which some of the entities are abstracted as parameters. A parameter can be, among other things, an algorithm, a representation scheme, or a sub-module. The design process is viewed as an instantiation of the various parameters comprising a generic architecture. However, the parameters can be fairly complex entities and the design task is non-trivial.

The structure of the generic architecture determines the basic solution strategy for solving the problem. For example, the continuous signal interpretation problem given earlier can be solved using a symbolic, knowledge based approach, or by statistical analysis of the data and the two solutions would have radically different architectures. A module description includes information about the input and output data flows of the module, the submodules/supermodules structural relations, the services it requires from other modules, the services it provides to an external module, the precondition and postconditions for each service provided by the module, and/or a program template that implements each service. The most interesting aspect of the module description is that some of its attributes are viewed as parameters of the module. Associated with each parameter attribute is a method which can be used to determine the value of the parameter. The complexity of the method depends on the type of the parameter. For example, it may be a simple process of selecting between a pre-determined list of alternatives, or it may involve sophisticated reasoning using domain knowledge and heuristic rules.

Continuous-Signal-Interpretation :Generic-problem
Signal-Inputs:)<var> : (SEQ :FROM <int> :TO <int> (<fields>)
<field-description>)]+
Body: WHILE <formula> DO <statements> ENDWHILE
Task Assumptions: <task-assumptions>
where
<fields> ::= <identifier> | <identifier> <fields>
<field-description> ::= EXIST <objects> SUCH-THAT <condition>
<statements> ::= <statement> ; <statements> | <statement>
<statement> ::= (IF <formula> THEN-DO <statement>) |
(FORALL <vars> <formula> DO <statement>) |
(PRINT <terms>)
<task-assumptions> ::= (UNRELIABLE-SIGNAL <var>) |
(REDUNDANT-SIGNAL <var>) |
(ASYNCHRONOUS-SIGNAL <var>) |

Fig. 2. Specification of the generic problem of continuous signal interpretation.

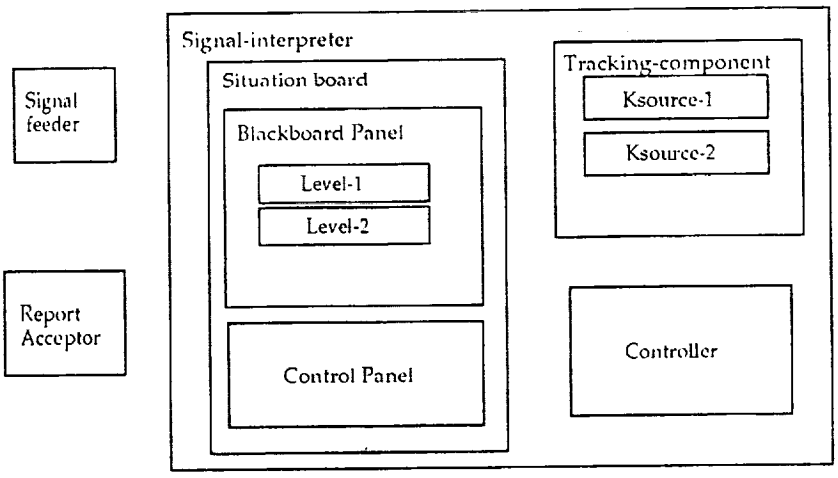


Fig. 3. A functional decomposition of the generic architecture for the continuous-signal interpretation problem. The architecture shows the main modules comprising the architecture.

Signal-Interpreter **isa**	module
submodules	Situation-board, Tracking-component, Controller
supermodule	CSI-system
inputs	?s : SEQ(signal)
outputs	?r : SEQ(report)
requires	(print-report ?r), (read-next-signal) :signal, (start-execution)
provides	(main)
calls	report-acceptor, signal-feeder
called-by	nil
parameters	
	1) Controller
	2) SituationBoard
constraints	
	1) Controller is instantiated to an EventDriven-Controller iff SituationBoard is instantiated to an EventDriven-SituationBoard.
	2) Only the TrackingComponent should have a dataflow into the SituationBoard.
	3) Only the Controller module can call the Tracking-Component.
	.....

Fig. 4. Representation of the Signal-interpreter module in the generic architecture.

Figure 3 shows the structural decomposition of a generic architecture for the continuous-signal-interpretation problem class and figure 4 shows a partial description of the signal-interpreter module of the generic architecture.

The domain model provides the ontology of terms and operations used to describe an application domain independent of a specific task; several different problems can be specified in a high-level language using this ontology. The primary components of the domain model are *objects* and *relations* between the objects. An object is an abstraction of some entity in the application domain, e.g., an aircraft or a signal. Associated with each object is a set of *attributes* which are properties that describe an instance of an object and *operations* that change the state of an object. The description of an operation includes pre- and post-conditions and optionally, a code template that implements the operation.

2.1 CUSTOMIZATION PROCESS

The customization process consists of refining a selected generic architecture into a detailed architectural specification based on the model of the domain and the problem specification. In KASE, the customization process is performed in an interactive and mixed-initiative setting. The role of KASE in the design process is that of an intelligent

design associate that provides suggestions on how to refine the architecture, carries out the commands invoked by the user, informs the designer of constraint violations in the design, keeps a record of the design steps and the dependencies between the steps so that incremental modifications to the design can be done efficiently.

The knowledge used by KASE in providing these kinds of assistance includes general, domain independent knowledge about software design, architecture-specific knowledge for the instantiation of various architectural parameters, as well as specific heuristic knowledge about design related to a particular domain. Most of the domain independent design knowledge is represented in the form of constraints (e.g. those relating different levels of a data flow diagram), and KASE contains mechanisms which automatically keep track of these constraints as well as heuristics for resolving constraint violations (Nii *et al.* 1991). The architecture specific knowledge includes a set of constraints governing the relationships between different components of the architecture, a library of reusable modules and schemas which can be used to instantiate the architectural parameters, and a collection of design rules and procedures that can be invoked by a designer to instantiate certain parameters and optimize the design.

To illustrate the customization process, consider the generic architecture shown in fig. 2. The parameters in the generic architecture include the following: 1) the submodules of the blackboard panel, 2) the type of information stored in the control panel, 3) the submodules of the tracking component, and 4) the scheduling and focusing strategies of the controller. Different instantiations of these parameters result in the creation of a widely different systems with different performances. KASE contains a set of design rules for instantiating these parameters, and a set of transformation rules that optimize the design (e.g. merging certain kinds of control signals into one for increased efficiency). The customization process for an implemented example in KASE is described in [Bhansali & Nii, 1992].

2.2 REDESIGN

Software design is characterized by frequent modifications either due to a design error or as a result of a change in the problem requirements or the computing environment. KASE uses different mechanisms to support these two kinds of modifications.

2.21 Redesign due to error in original design. KASE automatically checks for violations of several kinds of constraints and helps the designer modify the architecture to resolve the inconsistencies. The constraints in KASE are currently divided into three categories: 1) General architectural constraints (e.g. every data link must have a consumer and a producer); 2) Specific architectural constraints (e.g. there must be no data flow or control flow between submodules of the tracking component); and 3) Stylistic constraints that are derived from design principles that are considered 'good' (e.g a module must not be decomposed into more than *n* submodules at any level of abstraction).

Each constraint in KASE is associated with a *trigger*, a *predicate*, and an optional *resolving-action*. A trigger is a set of actions that can potentially cause the constraint to be violated, a *predicate* is a Lisp expression that checks to see whether the constraint is actually violated, and *resolving-action* is a set of actions that may be taken to remedy the constraint violation. KASE monitors the design activity and flags each constraint that is triggered by a user action. When a user indicates the completion of a design session, KASE checks the predicates for each flagged constraint to see whether the constraint is actually violated. Quite often, a constraint that gets violated by a design action is resolved by a later action, and such constraint violations should be, and are, transparent to the designer.

When KASE reports a constraint violation, the designer can ask KASE for a list of suggestions on how to resolve the error. Depending on the nature of the constraint, KASE presents a list of different actions that may be taken to remove the constraint violation. The user can then choose either one of the actions suggested by KASE or take some other action.

2.2.2 Redesign due to change in requirements. KASE provides tools that can help a designer in modifying parts of a design to meet new requirements without having to start from scratch. First, KASE maintains a history of all the design steps and allows the user to go back to any previous state of the design. It does this by replaying the design history from the initial state to the desired state.

A second redesign support provided by KASE is in localizing the effects of a design change. KASE uses dependencies between design steps to structure a linear design history into a lattice. When the user wants to undo the effect of a particular design step, KASE uses the position of that design step in the derivation history to determine what other design steps are affected by it [Bhansali, 1992].

3. Discussion

In this section we briefly discuss some of the issues, advantages, and limitations in our approach. One of the major issue in the design of domain-specific systems is concerned with acquiring and maintaining the extensive body of knowledge from multiple sources. This task, also known as domain modeling, is a manifestation of the classic

knowledge acquisition problem in expert systems. One way of viewing generic problems/tasks and architectures is to consider them as providing a skeletal knowledge base or shell which can be instantiated for different applications. Our long term goal is to provide a library of generic problems and associated architectures, which would provide a base from which various domain models can be instantiated.

A second issue is concerned with the flexibility of the resulting system. Domain specific systems utilize specialized design techniques which are well suited for a particular class of applications. However, since it is not possible to anticipate all subsequent changes in requirements, the specialised design techniques may not be adequate for extending the system beyond the original intended application. A major effort in the KASE project has, therefore, been expended in providing a domain--independent infrastructure which enables a user to modify an architecture through an integrated editor, pictorial and symbolic visualizations of the design from various perspectives, and a constraint maintenance subsystem that supports opportunistic design based on insights drawn from empirical studies of human designers [Guindon, 1990].

A third issue is concerned with the usefulness of the approach. Our approach involves a considerable investment in terms of building the initial knowledge structure, and we believe that the payoff is in being able to reuse generic architectures to design solutions for a family of problems. We need to identify such architectures and problem classes and use KASE for designing software systems for problems belonging to such problem classes.

The KASE system represents our initial attempt in building a prototype environment that can offer varying degrees of assistance to a software designer by employing diverse sources of knowledge. Our current work is focusing on extending the domain modeling representation to capture the dynamic behavior of a system by modeling states, transitions, events, and actions. We are also exploring the issue of design rationale capture and its reuse during redesign. KASE's current redesign capabilities were mentioned briefly in this paper. We are interested in extending these capabilities so that KASE can automatically incorporate certain changes in problem requirements into the design by using the design rationale.

Acknowledgements

The KASE system is a result of several people's work. We gratefully acknowledge the contributions made by Nelleke Aiello, Raymonde Guindon, Liam Peyton and Go Nakano who wrote most of the code for KASE.

References

Adelson, B. & Soloway, E. (1985). The role of domain experience in software design. *IEEE Transaction on Software Engineering*, SE-11(11):1351 - 1360.

Bhansali, S. (1992). Generic software architecture based redesign. AAAI Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse, Stanford, CA.

Bhansali, S. & Nii, H. P. (1992). KASE: An integrated environment for software design. *2nd International Conference on Artificial Intelligence in Design*, Pittsburgh, PA.

Brown, H. D., Schoen, E., & Delagi, B. A.(1986). An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures. Department of Computer Science, Stanford University, Technical Report STAN-CS-86-1136.

Graves, H. (1991). Lockheed Environment for Automatic Programming. 6th Knowledge-Based Software Engineering Conference, Syracuse, NY: 78-89.

Guindon, R. (1990). Designing the Design Process: Exploiting Opportunistic Thoughts. *Human-Computer Interaction*, 5:305-344.

Guindon, R. (1992). Requirements and design of DesignVision, an object-oriented graphical interface to an intelligent software design assistant. *ACM Proceedings of CHI'92*, Monterrey, CA.

Guindon, R., Krasner, H., & Curtis, B. (Eds.). (1987). *Breakdowns And Processes During The Early Activities Of Software Design By Professionals*. Ablex Publishing Corp.

Nii, H. P., Aiello, N., Bhansali, S., Guindon, R., & Peyton, L. (1991). Knowledge Assisted Software Engineering (KASE): An introduction and status June 1991. Knowledge Systems Laboratory, Computer Science Department, Stanford University, Technical Report KSL-91-28.

Nii, P.(1989). Blackboard Systems. In A. Barr, P. Cohen, & E. Feigenbaum (Eds.), *Handbook of Artificial Intelligence*. New York, NY: Addison-Wesley.

DOMAIN SPECIFIC SOFTWARE ARCHITECTURES -- COMMAND AND CONTROL

Christine Braun
William Hatch
Theodore Ruegsegger
GTE Federal Systems
15000 Conference Center Dr.
Chantilly, VA 22021

Bob Balzer
Martin Feather
Neil Goldman
Dave Wile
USC/Information Sciences Institute
Marina Del Rey, CA 90292

Abstract

GTE is the Command and Control contractor for the Domain Specific Software Architectures program. The objective of this program is to develop and demonstrate an architecture-driven, component-based capability for the automated generation of command and control (C2) applications. Such a capability will significantly reduce the cost of C2 application development and will lead to improved system quality and reliability through the use of proven architectures and components.

A major focus of GTE's approach is the automated generation of application components in particular subdomains. Our initial work in this area has concentrated in the message handling subdomain; we have defined and prototyped an approach that can automate one of the most software-intensive parts of C2 systems development.

This paper provides an overview of the GTE team's DSSA approach and then presents our work on automated support for message processing.

The DSSA Concept

DSSA is based on the concept of an accepted generic software architecture for the target domain. As defined by DSSA, a software architecture describes the topology of software components, specifies the component interfaces, and identifies computational models associated with those components. The architecture must apply to a wide range of systems in the chosen domain; thus it must be general and flexible. It must be established with the consensus of practitioners in the domain.

Once an architecture is established, components that conform to the architecture—i.e., that implement elements of its functionality in conformance with its interfaces—will be acquired. They may be acquired by identifying and modifying (if required) existing components or by specifically creating them. One of the ways they may be created is through automated component generation. DARPA has sponsored work in this area at USC Information Sciences Institute -- the AP5 application generator project, and is interested in incorporating this or related technology.

The existence of a domain-specific architecture and conformant component base will dictate a significantly different approach to software application development. The developer will not wait until detailed design or implementation to search for reuse opportunities; instead, he/she will be driven by the architecture throughout. The architecture and component base will help define requirements and allow construction of rapid prototypes. Design will use the architecture as a starting point. Design and development tools will be automated to "walk through" the architecture and assist the developer in the selection of appropriate components. The ultimate goal is to significantly automate the generation of applications. A major DSSA task is to define such a software lifecycle model and to prototype a supporting toolset.

These activities will be accompanied by extensive interaction with the development community for the target domain, and by technology transition activities. One aspect of this is that each domain team is working closely with a DoD agency that carries out major developments in the designated area. The GTE team is working with the US Army Communications and Electronics Command.

Why Command and Control?

There are many reasons why the command and control domain is an excellent target for DSSA technology. It is a high payoff area; command and control systems are needed even in the current military climate. (This is particularly true when one recognizes that applications such as drug interdiction fall within the C2 "umbrella".) It is a well-understood area; most of the processing performed in C2

applications is not algorithmically complex. However, C2 applications are very large, and much of this size comes from repeated similar processing -- for example, parsing hundreds of types of messages. In addition to this commonality within applications, there is much commonality across applications. Multiple C2 systems must handle the same message types, display the same kinds of world maps, etc.

The kinds of commonality in C2 applications are very well-suited to DSSA techniques. In some areas, components can be reused identically; these can be placed in the DSSA component base and highly optimized. In other areas, components will be very similar in nature but differ in the particulars, e.g., message parsing. These areas are a natural fit to the DSSA component generation technology, allowing a table-driven generator to quickly create the needed specific component instances.

GTE's Approach

Figure 1 illustrates GTE's overall approach to the DSSA program.

Initially, project work will follow two parallel threads. The first will define a software process model appropriate to architecture-driven software development and will develop a toolset to support that process. The second will establish a capability that implements the process for the command and control domain, based on a C2 architecture and a set of reusable C2 components.

The DSSA process model will address all aspects of the software life cycle. It will describe activities for establishing system requirements, developing the software system, and sustaining the system after delivery. The DSSA toolset will support all of these activities, automating them as far as possible. In particular, it will automate system development activities by using the architecture as a template, guiding the selection of available reusable components, and automating the generation of specific required components. The toolset will be constructed insofar as possible from available tools -- both commercial products and products of the research community. In particular, it will make use of USC/ISI's AP5 application generator, DARPA/ STARS reuse libraries, and DARPA/Prototech tools. Open tool interfaces will be emphasized to minimize specific tool dependencies, thus making the toolset usable in the widest range of environments.

Fundamental to the C2 DSSA capability is the development

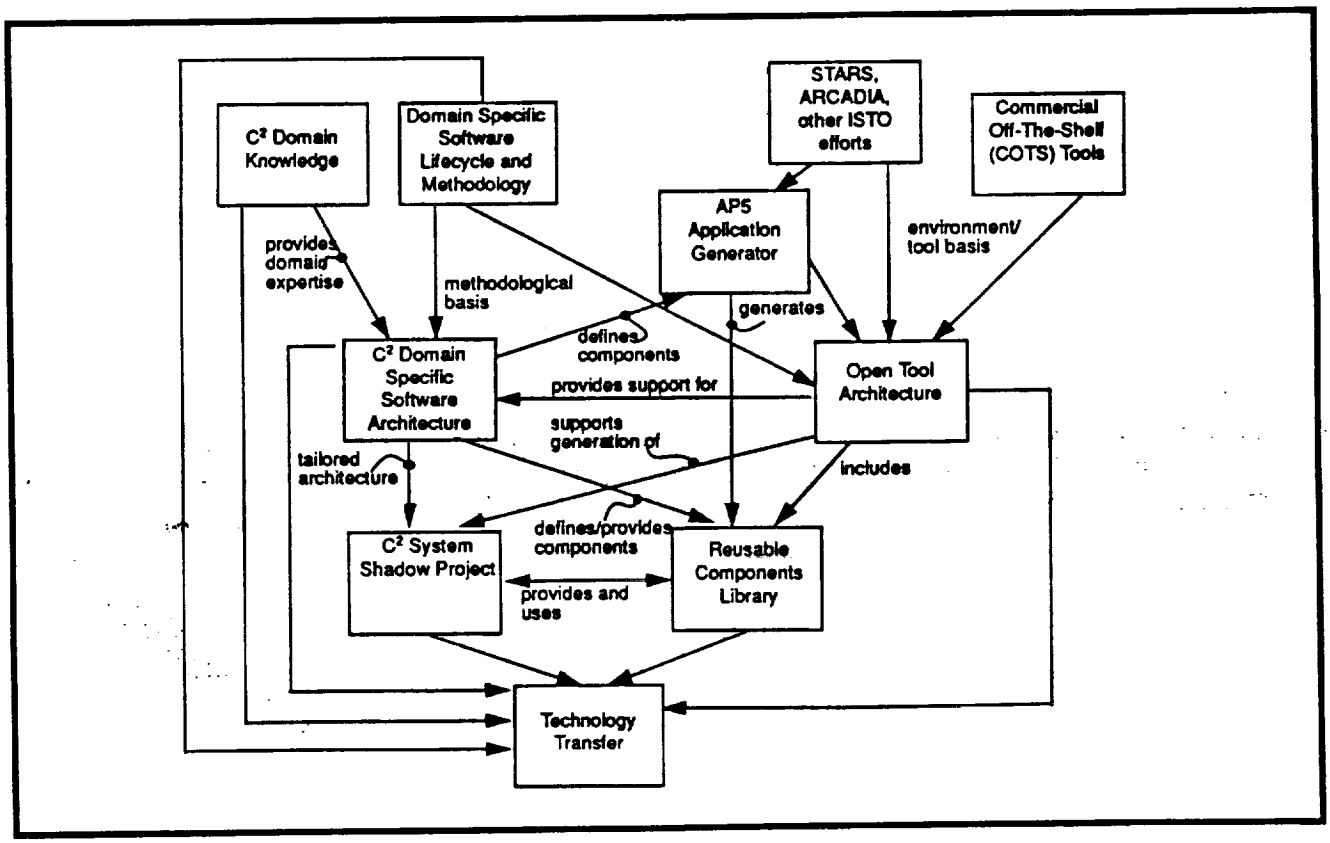


Figure 1. GTE's DSSA Approach

of a C2 software architecture. This starts with development of a multi-viewpoint domain model, created through interaction with all elements of the DoD C2 community. The automated Requirements Driven Development (RDD) methodology will be used in model creation. From this, an object-oriented software architecture will be developed. The architecture will tie back to the multi-viewpoint model so that mappings to different views of the domain functional decomposition are apparent. George Mason University's Center for C3I will play a major part in this modeling and consensus-building activity. A base of components conforming to the architecture will then be developed. Many of these will be existing components, perhaps modified to fit the architecture. Others will be automatically generated using AP5.

The DSSA capability will be demonstrated by development of a prototype C2 system, most likely an element of the Army Tactical Command and Control System (ATCCS). An independent metrics/validation task will assess the effectiveness of the approach and gather metrics. The methodology and toolset will be revised based on findings and further necessary research will be identified.

Throughout the program, a technology transfer task will present results in conferences, papers, seminars, and short courses. The George Mason University Center for C3I will serve as a focal point for technology transfer.

Application Generation

The Technology

Application generators are tools that permit software developers to create software application programs in a much higher-level language tailored to the application domain. These programs are automatically translated by the application generator to a lower-level language, thus "generating applications." This greatly reduces the effort required to create working applications, typically by at least an order of magnitude. The benefits are analogous to those achieved by moving from assembly language development to use of standard procedural languages such as FORTRAN, C, and Ada.

Fourth Generation Languages (4GLs) are application generators for DBMS-oriented information system applications. Because 4GLs focus on a narrow class of applications, they can include very powerful constructs that allow software to be developed quickly and easily by those familiar with the application domain. Management Information System (MIS) developers using 4GLs achieve productivity improvements of as much as 50-100 times over traditional (usually COBOL) language users.

Application generators can be (and have been) developed for other types of applications as well. They are best suited to narrow domains, or subdomains of large domains such as C2. Because they require a domain specific vocabulary for expressing applications, they are generally unique to the domain or subdomain and not easily modified to handle other domains. Creation of an application generator for a particular domain, furthermore, is a significant undertaking. Development of an application generator is most appropriate in domains that are well-understood and in which many different developments perform primarily the same kinds of processing.

The AP5 Approach

USC Information Sciences Institute (ISI) has developed a capability (called AP5) that supports the development of application generators. AP5 is based on the concept of *relational abstraction.* The application developer identifies abstract data objects and the logical relationship among them. Effectively, the developer has access to a "virtual database" expressed succinctly in terms of the known structure of the domain's data model. Application behavior is then expressed in terms of these data objects, accessing them associatively via queries and modifying them based on values of other objects. This allows the user to concentrate on behavior rather than representation, and provides the power to express that behavior at a very high level.

Providing an AP5 application generator for a particular subdomain requires the development of a domain-specific language for that domain. This is a relatively straightforward task because the language, regardless of domain, involves the same fairly simple set of relation-oriented constructs for expressing data relationships, validations, and actions. It is also a critical task, because the expressive capability of this language is what provides the application generator's power. A translator is then developed to map the language to an underlying program generator, which produces executable procedural code. This is also not too complex, as all languages contain similar constructs. Most of the work is done by the underlying generator. (Currently the system generates LISP; an Ada generator is in development.)

A drawback to many existing application generators is poor efficiency of the generated code. This has, in many cases, made these generators suitable only for developing prototypes. AP5 addresses this problem by allowing the user to specify *annotations* that provide guidance to the translator on desired implementations of specific operations. These annotations can be added incrementally while tuning to achieve desired performance.

AP5 can play a key role in the C2 DSSA program. We anticipate that a number of C2 subdomains will be amenable to this approach. By developing generators for those subdomains we can achieve two major advances in productivity:

- DSSA users can use the generators to create specific components in the subdomain with far less effort.
- DSSA architects can use the generators to create reusable subsystems that can then form part of the component base available to DSSA users.

We have already identified the message handling subdomain as a candidate for AP5 technology; a tentative choice for the next area to tackle is fusion processing.

Figure 2 shows the activity flow that will be followed: identifying classes of components (subdomains) to be addressed, based on the architecture; defining domain specific languages and producing generators; developing annotations to permit optimization; and generating reusable application components.

C2 Message Handling

As indicated in Figure 3, the message handling subsystem is one of the key interfaces between a C2 system and the "outside world". It provides a means of communicating information between different C2 systems and to/from other C2 resources (such as vehicles and weapon installations). Messages may be text or bit streams; we will deal here with text messages. Some text messages are free-form, but most today follow standard prescribed formats; we will deal with formatted messages.

C2 messages are created by humans (on the transmitting side of the interface) according to a written description of the formats. The receiving side parses the message (according to an encoded understanding of the standard format), validates it for correctness, and places the received information in the database for use by other parts of the system (for example, decision support).

There are several standard families of messages, for example NATO and JINTACCS messages. Each of these can include several hundred message types; for example, there are approximately 300 NATO message types. (Many types of messages are shared by several message families.) Message formats are described in massive documents using *ad hoc*, non-standard description methods. Typically the descriptions involve much prose. For example, Figure 4 shows the description for a single line in one type of message. Furthermore, it is not a complete description; many field descriptions cross-reference to other descriptions.

A message consists of a number of such lines (called *datasets*— may be more than one physical line) grouped together in an *envelope* (which contains from/to information, classification level, etc.). While each type of message can contain only certain kinds of datasets, many are optional and their order is generally not prescribed (though there are exceptions). Validity of datasets can depend on other datasets in the message. Each dataset contains a prescribed sequence of *fields*, separated by slashes, with a required order and a well- defined format. Field validity can depend on values in other fields of that dataset as well as in other datasets in the message. Figure 5 is an example message (excluding the envelope).

The code involved in writing the software to implement message handling is extensive and error prone. Working from the prose specification, programmers write code to extract each field from each dataset, validate it according to the specified rules, translate it to the appropriate internal representation, build database update transactions, and write to the database. Typically, a single message type can take from 5000 - 100,000 lines of HOL code. The Navy WWMCCS system uses approximately 4 million lines of code to implement 30 message types. Clearly this is a part of C2 system development that should be considered for automation.

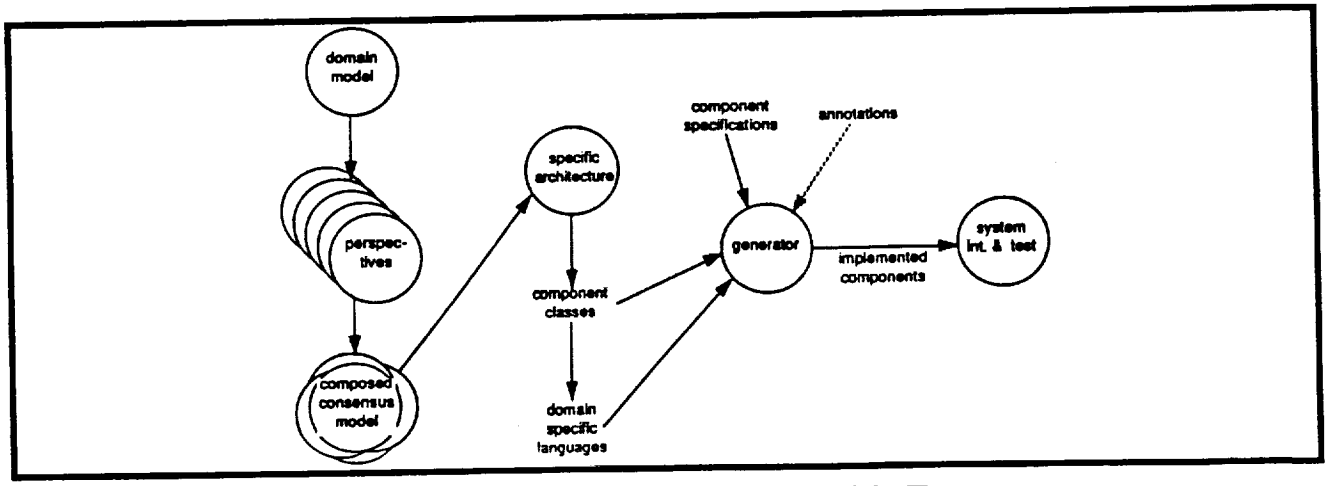


Figure 2. DSSA Application Generation Activity Flow

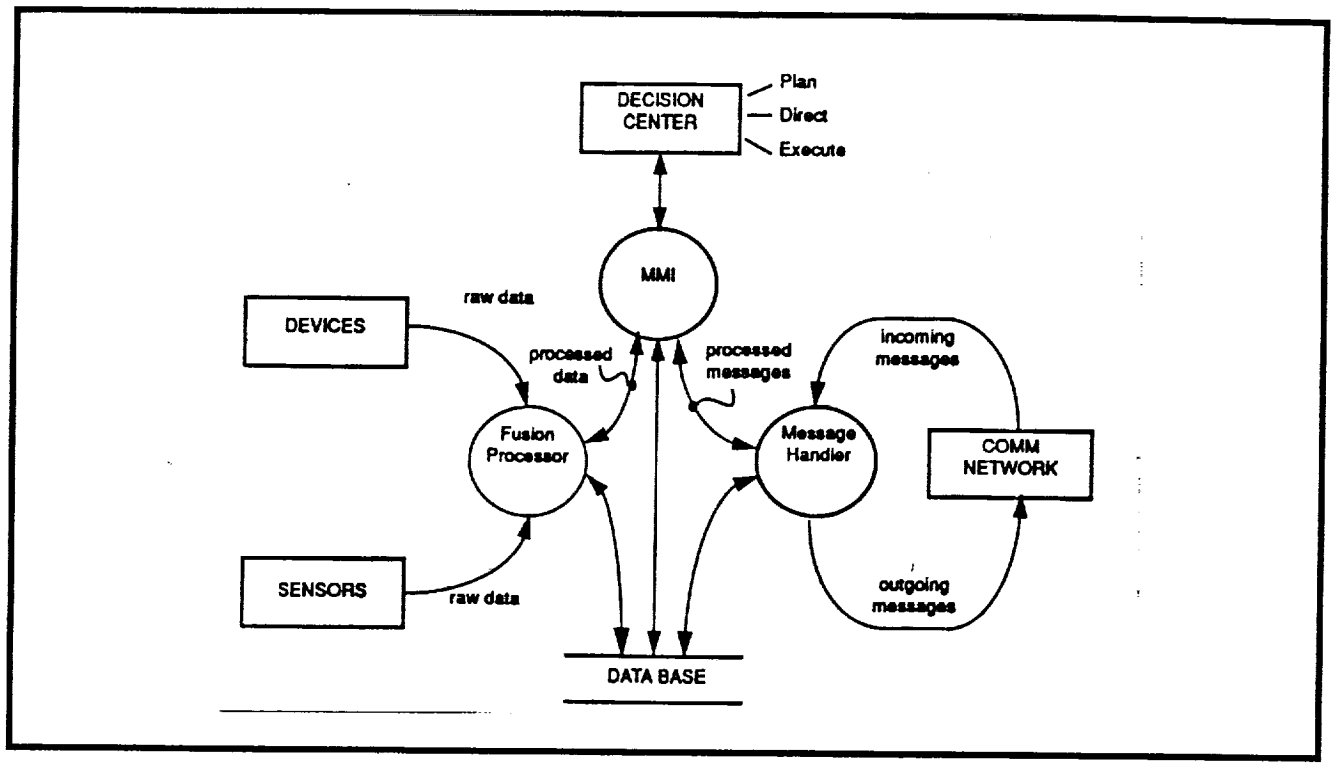


Figure 3. C2 System Operations

Automating C2 Message Handling Using AP5

To automate C2 message handling using AP5, we have developed a language specific to the message handling subdomain that provides constructs for specifying message formats, for indicating required validations, and for describing desired database updates.

Specifying Message Formats

Message formats are described in a simple set language that indicates which datasets are allowed and which are optional for a particular message type. For example,

```
type SPOT = (FORCE), (SHIPTK | AIRTK | AIRCRAFT),
            SHIP
```

would indicate that a SPOT message consists of an optional FORCE dataset, an optional occurrence of one of the SHIPTK, AIRTK, or AIRCRAFT datasets, and a required SHIP dataset.

Message format descriptions can be accompanied by *validations* that indicate which combinations of datasets are valid. For example,

```
type SPOT = (FORCE), (SHIPTK | AIRTK | AIRCRAFT),
            SHIP
  validations
    disallow MSGID.message-serial-number;
    require SHIP.location
            no SHIPTK and no AIRTK requires FORCE;
```

indicates that the message-serial-number field of the MSGID dataset must not be present, the location field of the SHIP dataset must be present, and, if no SHIPTK dataset and no AIRTK dataset is present, the FORCE dataset must be present.

Specifying Datasets

Dataset formats are described in terms of the fields that make up the dataset and the format of each of those fields. Fields are ordered, so each dataset is characterized by a sequence of fields. Optional fields are indicated by parenthesizing them. Mutually exclusive fields are indicated by alternative bars. As for message formats, dataset descriptions can include validations. For example, a dataset description of a MSGID dataset might be:

```
dataset MSGID = message-code-name (originator)
                (message-serial-number) (as-of-month)
                (as-of-year) (as-of-DTG)
  validations
    as-of-DTG precludes as-of-month;
    as-of-DTG precludes as-of-year;
    as-of-year requires as-of-month;
    message-code-name /= SPOT requires originator;
    message-serial-number and no as-of-DTG
            requires as-of-month;
  field message-code-name = A*26;
```

Data Set ID: MSGID

Fld	Element Descriptive Name	Descript.	M	Edit Rule	Remarks
1	Message Code Name	25 AN	x	1. Must be a member of the approved set of message code words.	
2.	Originator	25 AN	x	1. Must be a plain language address or approved short title	a. Plain language addresses are validated against values found in the references
3.	Message Serial Number	3 N	a	1. Positive integer between the values 001 to 999.	a. May be required for specific messages.
				2. Out of sequence may indicate missing message. See rules for specific msg. code word.	b. Sequence is restarted on 1 Jan each year. May be rolled over when upper limit is reached. c. For Command authorities serial may be validated to maintain order when processing reports.
4.	As-of-Month	3 AN	a	1. Standard abbreviation for month message sent.	a. Required if serial number is used and as-of-DTG not present. b. Not allowed if as-of-DTG not present.
5.	As-of-Year	4 N	o	1. May not be a future year.	a. As-of-Month must be present.

Figure 4. Example Message Line Description

```
NATOUNCLASSIFIED
SIC: NSR
EXER /OPEN GATE 91//
MSGID /NAVSITREP/CINCIBERLANT/135/DEC/91//
PART /I/HOSTILE//
FORCE /OR523/3/37000N0-012000W3/145/17K/H//
SHIP /OR523A/KARA/-/CG/-/UR//
SHIP /OR523B/KRESTA//
SHIP /OR523C/KRESTA//
SUBTK /OR734/33000N6-010000W1/095/9K/M//
SUB /OR734/TANGO//
PART /II/UNKNOWN/NC//
PART /III/FRIENDLY//
FORCE /CTU 405.1.2/5/420015N2-1333440W8/175/20K//
FORCE /CTU 387.3.2/2/36010N0-004380W5/090/5K//
AMPN /MINE SWEEPING GROUP...//
AIRTK /934/33000N6-010000W1//
AMPN /ONE P-3 SEARCHIN BOX...//
```

Figure 5. Example Formatted Message

```
field originator = A*25;
field message-serial-number = N 3;
field as-of-month - month;
field as-of-year = N 4;
-- as-of-DTG in form: DDHHMMZS      MMMYY
field as-of-DGT - day, hour, minute, (Z), SUM1, month,
          year;
field SUM1 = N 1;
field day = N2;
field hour = N 2;
field minute = N 2;
field month = A 3;
field year = N 2;
```

Specifying Database Transactions

The C2 message description language also includes a means for describing the transactions to be carried out for each received message. An example of a segment of such a specification is:

```
{insert msg_Orig_Sr (ORIGINATOR = PROSIGN.FN,
   MSG_TYPE = MSGID.Code,
   MSG_DTG = sortable_date (ENVELOPE.DTG),
   CLASSIFY = classification_code(ENVELOPE.Sec));
   ...
```

The database update language also includes tests of field values, so that updates can be conditional on those values, and a capability to allow a sequence of updates to be named and reused in other update instructions. This simple language provides all the power needed to describe the database transactions resulting from received messages.

Implications

Clearly, automated generation of message handling software can save greatly on the labor involved in creating such software. A message handling subsystem that requires 4 million lines of HOL code should require less than 1% of that in the message description language.

Perhaps more significantly, there will be little reason to write most of the code more than once. The code required to parse and validate a message of a particular type is not specific to the system being implemented. Once the message specification is developed in the message description language, it can be reused. Minor changes in the specification of required database updates can be easily implemented for individual systems.

An even more far-reaching impact of this work is the development of a precise, unambiguous way of describing message formats. Rather than the *ad hoc* prose descriptions now used in describing message formats, the message description language can be used directly. This will eliminate errors in understanding and correctly implementing message descriptions.

This precise message description mechanism, along with the built-in incentive to reuse message description implementations, will contribute substantially to the development of more error-free message handling subsystems. A major aspect of this benefit is improved interoperability, as systems will no longer be dependent on the programmers' understanding of message formats. All implementations will share a common understanding and be able to interoperate with the full power and precision envisioned for formatted messages.

Acknowledgment

The work described in this paper has been supported by the Defense Advance Research Projects Agency through U.S. Army Communications-Electronics Command Contract No. DAAB07-92-C-Q502 and through NASA Ames Research Center Contract No. NCC 2-520.

References

[1] Balzer, Bob and Martin Feather, Neil Goldman, Dave Wile, "Proposal for DS Languages for C3 Messages," USC/ISI working paper, 1992.

[2] Braun, Christine L. and William L. Hatch, "Software Reuse Through CCIS Architecture Standardization," *Proceedings of the 11th AFCEA Europe Symposium and Exposition*, October 1990.

[3] Hatch, William, "Example Message Descriptions and Database Transactions," GTE working paper, 1992.

[4] Ruegsegger, Theodore, "Domain Specific Software Architectures -- Command and Control," briefing slides, CECOM Real-Time/Reuse Technical Interchange Meeting, Ft. Monmouth, NJ, February 1992.

[5] Wile, David S., "Adding Relational Abstractions to Programming Languages," *Proceedings of workshop on Formal Methods in Software Engineering*, Napa Valley, CA, May 1990.

[6] Balzer, Robert, "A 15 Year Perspective On Automatic Programming," *IEEE Transactions on Software Engineering*, Nov. 1985

[7] Cohen, Donald, "Compiling Complex Database Triggers," *Proceedings of 1989 ACM SIGMOD* (1989), ACM

[8] Goldman, Neil and K. Narayanaswamy, "Software Evolution through Iterative Prototyping," to appear in the *Proceedings of the 14th ICSE Conference*, IEEE, Melbourne Australia 1992.

Issues in Knowledge Representation to Support Maintainability: A Case Study in Scientific Data Preparation

Steve Chien, R. Kirk Kandt, Joseph Roden and Scott Burleigh

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099

Todd King and Steve Joy

Institute of Geophysics and Planetary Physics
University of California at Los Angeles
Los Angeles, CA 90024-1406

Abstract

Scientific data preparation is the process of extracting usable scientific data from raw instrument data. This task involves noise detection (and subsequent noise classification and flagging or removal), extracting data from compressed forms, and construction of derivative or aggregate data (e.g. spectral densities or running averages).

A software system called PIPE provides intelligent assistance to users developing scientific data preparation plans using a programming language called Master Plumber. PIPE provides this assistance capability by using a process description to create a dependency model of the scientific data preparation plan. This dependency model can then be used to verify syntactic and semantic constraints on processing steps to perform limited plan validation. PIPE also provides capabilities for using this model to assist in debugging faulty data preparation plans. In this case, the process model is used to focus the developer's attention upon those processing steps and data elements that were used in computing the faulty output values. Finally, the dependency model of a plan can be used to perform plan optimization and runtime estimation. These capabilities allow scientists to spend less time developing data preparation procedures and more time on scientific analysis tasks.

Because the scientific data processing modules (called fittings) evolve to match scientists' needs, issues regarding maintainability are of prime importance in PIPE. This paper describes the PIPE system and describes how issues in maintainability affected the knowledge representation used in PIPE to capture knowledge about the behavior of fittings.

Introduction

Scientific data preparation is defined as the application of multiple transformations to collected data sets in order to produce data in an easily usable form. The questions a scientist asks dictate which data are to be collected as well as which transformations are to be applied. The need for simplified scientific data preparation has increased due to the volume of data now collected and the diverse uses for any specific type of data. Automated scientific data processing systems can be used to simplify this process.

While general scientific data processing systems have existed for some time, the complexity of data types and transformations required in specific domains renders these systems of limited utility. As a result, many scientific teams develop their own software systems to accomplish the data preparation required in their specific domain. These systems suffer because they become too specific, and the effort spent developing such systems are only of value within the context of a particular domain and task. Because scientists desire to reuse their work, hybrid systems are appearing which provide useful analysis tools and definition of domain-specific data types and transformations. Plans are developed in these systems which specify which of the transformations to apply to a collection of data sets. By the nature of the processing steps required in many domains, these plans can become quite complex. We are now at a point where the complexity of these tools requires significant expert knowledge to use.

Master Plumber [King & Walker 1991] is a software tool developed by the UCLA Institute of Geophysics and Planetary Physics to create programs to prepare scientific data. While its primary area of application has been time-series magnetometer data, the tool is applicable to the general task of scientific data preparation.

Master Plumber is a dataflow system. Thus, in Master Plumber, data elements are represented by columns, which are streams of data being processed as they move through the system. Data processing steps are called fittings, and a

plan to process a particular form of a dataset into another form is called a blueprint.

Thus, as shown in Figure 1, raw data might be read in using an intro_flatfile fitting, a running average computed using a runstat fitting, and the results written into an output file.

```
1.  intro_flatfile infile=foo
      columns=bx
2.  runstat length=1287 shift=1
      columns=bx
3.  write_flatfile outfile=bar
      columns=bx,rabx overwrite=YES
```

Figure 1: A Simple Blueprint

A major difficulty in constructing blueprints is tracking the many fitting and column interactions. While a typical blueprint might use 25 columns and 20 fittings, the more complex blueprints use hundreds of columns and 30 or more fittings. Because of the number of possible interactions, constructing and debugging scientific data preparation blueprints is a time-consuming task requiring expert knowledge.

Because of the complexity of the data preparation task, users sometimes make errors in blueprint construction. One type of construction error occurs when a user forgets to set up the data needed for a particular step. Unfortunately, this type of error can go unnoticed until far into the execution of the blueprint, wasting valuable time.

Another common situation is that the exact method of processing the data is dependent upon the character of the data. In this case the user will use some default methods for processing the data, examine the results, and modify the options. This tuning cycle continues until the data is in a satisfactory form.

The final aspect of blueprint development which complicates the development process is that new fittings are added to a system as new needs and requirements arise. In addition, new fittings also evolve with new options and characteristics being added. Any intelligent tool must be readily changed to remain useful in such a dynamic environment.

Currently there are approximately 65 fittings which are part of the standard Master Plumber system. These fittings perform a variety of transformations on the data flow, such as: introducing and writing data into several formats; displaying data on the screen; and actual numerical transformations. There are support libraries which allow for fittings to be written in either C or FORTRAN. A special fitting called PLISP takes programs written in a C-like language and performs the transformations on the data flow. This allows for new processing steps to be initially tested as PLISP programs and later be integrated as full-fledged fittings into the Master Plumber system.

Some scientists use data preparation systems indirectly with the help of software support personnel who write and debug the actual data preparation plans. The goal of PIPE is to make Master Plumber easy enough to use such that this type of support is not necessary. The combination of PIPE and Master Plumber will allow the blueprint developer to develop blueprints easier and faster, allowing them to spend more time on data analysis and less time on data preparation.

Overview

To achieve these goals of assistance in the scientific data preparation process, PIPE [Chien et al. 1992] provides four capabilities:

1. constraint checking to detect invalid blueprints before execution;
2. diagnosis assistance of blueprints through dependency analysis;
3. optimization of blueprints through dependency analysis; and ,
4. runtime estimation, using models of fitting runtime performance.

The architecture of the PIPE system is shown in Figure 2. PIPE accepts a blueprint file and a set of descriptors for datafiles and uses a fittings knowledge base to construct a dependency graph representing the computations to be performed by each of the fittings in the blueprint. This blueprint parsing phase uses knowledge of fittings and their options to construct a dependency graph, which indicates for each fitting which columns are accessed and used to modify existing columns, create new columns, or remove existing columns. This dependency graph can then be used by the constraint checking module which determines if any of the constraints associated with the fittings have been violated.

In cases where blueprints must be debugged, PIPE can use the dependency graph to support isolation of the fault in the blueprint. Because the dependency graph tracks all of the operations upon the columns, when the user detects an error in one of the output columns, PIPE can present a list of fittings which modified the column in question. The user can then focus his attention upon these fittings, to determine where the error was introduced into the data, sometimes by plotting intermediate data. After isolating the first fitting at which the column is faulty, the user can query PIPE for information on the fitting to determine which columns were used to compute the changed column. This process continues until the fault is isolated to the data, fitting option settings, or fitting code itself.

PIPE also provides an optimization capability. Because PIPE constructs a full computation dependency graph, PIPE can determine the last fitting in which each column of data is used in the blueprint. Thus unneeded data can be removed from the dataflow, decreasing the execution time Because many fittings operate on data by default, PIPE distinguishes between default processing and explicit

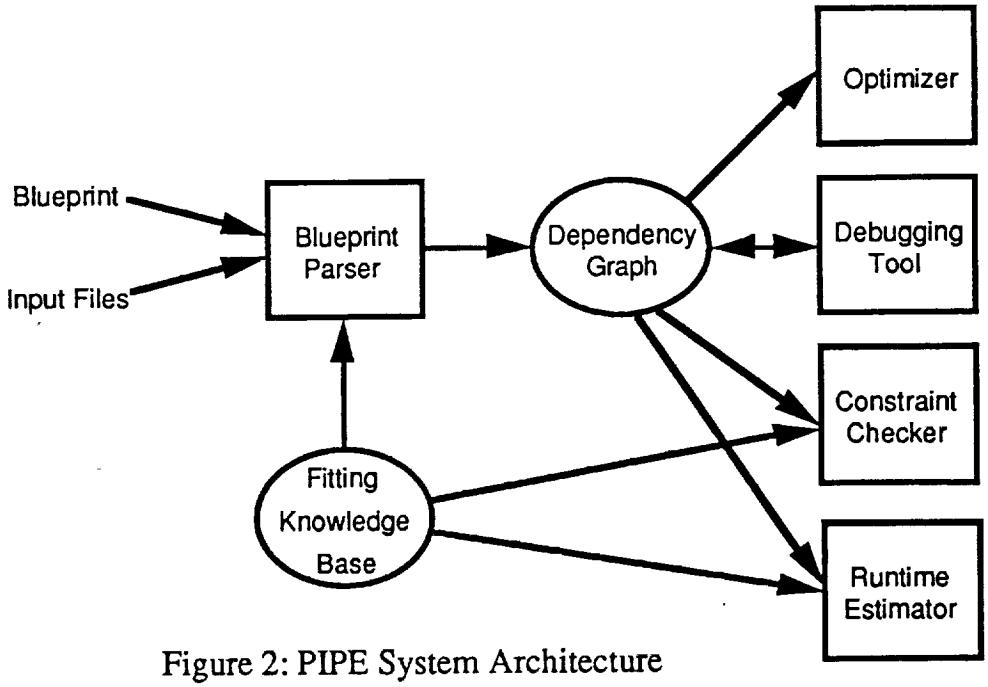


Figure 2: PIPE System Architecture

processing. Default computation which does not result in a program output (e.g. plot, output file) can also be removed.

Finally, PIPE provides a runtime estimation capability. Using the dependency graph to determine which columns each fitting processes, and models of runtime for each fitting type, PIPE can provide an estimate of how long the blueprint will take to run to completion for the specified datafiles.

Blueprint Parsing

In order to provide assistance in blueprint development, PIPE constructs a dependency network representation of a blueprint. When a blueprint is read in by PIPE, it is processed from the first step onward. For each fitting, PIPE uses:

- methods stored in the fitting knowledge base,
- default values stored in the fittings knowledge base,
- fitting options,
- a list of existing columns in the flow, and possibly
- an input file

to determine:

- any new columns created by the fitting,
- any existing columns modified by the fitting,
- existing columns deleted by the fittings.

Additionally, for any new or modified columns, PIPE determines:

- the set of columns accessed in computing the value for the column.

Because columns may be processed by default or explicitly selected, the dependency network also makes note of this distinction. This facet of the processing is important in order to take appropriate action when optimizing the blueprint (see below).

Constraint Checking

Constraint checking occurs while the blueprint file is being parsed (i.e., prior to execution). A description of the constraint checking algorithm follows.

```
During Parsing
   for each fitting in the blueprint
      for each option specified
         check option type constraints
      check for required options

After Parsing
   for each parsed fitting in blueprint
      for each option in fitting
         check option value constraints
      check inter-option constraints
      check dependency constraints
   check inter-fitting constraints
```

Diagnosis Assistance

PIPE also provides a blueprint diagnosis facility. This capability supports two basic types of queries: column-centered queries and fitting-centered queries. The column-centered queries are of the form

```
"What fittings affected <column>
before <fitting>?"
```

and default to the entire blueprint. This question can be easily answered using information from the dependency network. PIPE steps through the fittings in the blueprint and determines those fittings which create, modify, or delete <column>. This list of fittings is then displayed to the user in graphical form. The fitting centered queries are of the form

```
"What columns did <fitting>
affect?", and

"What columns did <fitting> access
in performing its processing to
affect these columns?"
```

These types of queries can be answered by interpreting the dependency graph information on the designated fitting. The first query can be answered by determining the set of columns created, modified or deleted by the fitting. The second query can be answered by accessing dependency network information regarding which columns were accessed by the fitting in performing these operations.

Blueprint Optimization

PIPE also provides a limited blueprint optimization capability. In this capability, PIPE examines the dependency graph of each column and determines the last fitting at which each column is accessed explicitly (i.e., not by default). PIPE then recommends removing this column immediately after this fitting. If this column is not processed in the remainder of the blueprint, this removal does not significantly alter the runtime of the blueprint. However, many of the fittings process all of the columns in the flow by default. Thus, when a column that is processed in the remainder of the blueprint is removed from the data flow a significant speedup can result. While commonly used blueprints are likely to have unused columns optimized by hand, automating this process relieves the user of the burden of determining the point at which a column can be removed. Additionally, by allowing PIPE to automatically determine the correct places to remove columns, PIPE reduces the chance that a user will inadvertently prematurely remove a column from the data flow, which would cause an error.

Runtime Estimation

The final capability that PIPE provides is runtime estimation. PIPE estimates the runtime of a blueprint for a specific data set by applying the following algorithm:

```
for each fitting in the blueprint
   identify fitting runtime model
   compute runtime given dataset size
   add runtime to total runtime
   compute new size of dataset
```

Tracking the size of a dataset in Master Plumber can be a difficult task. Original data set sizes are determined from input files. When data of different temporal granularity are introduced into an existing flow, or when decimation operations are performed, data set sizes will need to be recomputed. Sometimes a fitting can affect the size of the dataset in a manner that depends on the exact data processed. In these cases, the exact dataset size cannot be determined, so PIPE estimates the size of the dataset at the output of the fitting. These estimations are sufficient for giving the user reasonably accurate runtime estimates.

An Example

We now illustrate each of the capabilities of PIPE using example blueprints. For an example of constraint checking, suppose a user has created a blueprint containing the following statement:

```
4. bin columns=bx delta=60.0 min_max
```

Because the option `min_max` requires that a value be specified, PIPE would indicate a constraint error such as:

- `Fitting 4. bin option min_max required value not found; string type required.`

As another example of the constraint checking, consider the following blueprint statement:

```
7. crossavg except=time avgname=xavg
```

Assuming the user removed the column named `time` earlier in the data flow, PIPE would issue a constraint error indicating:

- `Fitting 7. crossavg option except undefined column time; a column with that name was deleted at fitting 4. drano.`

An example of the diagnosis capability supported by PIPE is illustrated in the following scenario. Figure 3 shows a Master Plumber blueprint file. Suppose that the user

examines the output of the blueprint and determines that column o2 is producing results that are incorrect. The user tries to determine what may have affected column o2 by querying PIPE:

```
Q: Which fittings created or
   modified column o2?

A: Fitting 10. drano created column
   o2.
   Fitting 12. plisp modified column
   o2.
```

The user determines that the o2 column was still incorrect before fitting 12. plisp, so the user wants to determine what columns were accessed by and were used in creating o2.

```
Q: Which columns were accessed by
   fitting 10. drano in order to
   create column o2?

A: Column raraby was accessed by
   fitting 10. drano in order to
   create column o2.
```

The user then continues backtracking through the blueprint to isolate the error:

```
Q: What fittings before fitting 10.
   drano modified column raraby?

A: Fitting 9. runstat created and
   modified column raraby.
```

By using PIPE in this way, the user can focus his attention directly upon the possibly faulty fittings instead of having to examine every fitting and column.

PIPE also uses the dependency graph to optimize blueprints. Because PIPE can determine which fittings modify which columns in the blueprint, PIPE can determine the last point at which each column is needed in the blueprint. In the example blueprint shown in Figure 3, PIPE makes the following recommendations for removal:

```
never introduce column rim

remove sens_x, senx_y, sens_z and bz
   after fitting 4

remove bx, by after fitting 8

remove rabx, raby after fitting 9

remove bxc, byc, bzc, and stime
   after fitting 12
```

PIPE also provides runtime estimation capabilities. For the optimization example shown above, PIPE estimates that the non-optimized blueprint will take 11:32 +/- 1:04 to run and the optimized blueprint will take 9:58 +/- 0:58 to run.

Issues in Design for Maintainability

The central concern in the PIPE knowledge representation was that the PIPE knowledge base be easy to maintain. While this is a concern in any knowledge-based system, it was particularly important in PIPE because fittings capabilities, options, and defaults, evolve because of changing scientists' needs. The majority of the knowledge represented in PIPE is used for the pre-runtime constraint checking. Thus, we focussed upon ensuring that these constraints be in a form that requires minimal change when fittings are changed.

In order to be easily maintainable, fitting constraints are implemented in three ways. First, basic option requirements constraints and argument requirements are specified in a simple language. This specification is then combined with a translator to generate C code which checks the options and option values against type and option requirement constraints. For example, each option for a fitting may be optional, or required (e.g., all fitting of this type must have this option specified) or be allowed to appear multiple times. Additionally, for each option arguments have associated constraints (e.g., all occurrences of this option must have an argument specified with the option). This structure affects maintainability as follows. When a change to a fitting is made which affects this information, the specification must be changed in the fitting knowledge base file. A translator is then used to automatically regenerate the associated constraint checking code so that the future constraint checking corresponds to the updated fitting.

The second type of constraint are simple, commonly occuring constraints, such as range constraints and inter-option range constraints (e.g., the value of option 1 must be greater than the value of option 2). These constraints are represented in a simple constraint language and stored in the fitting knowledge base file. When the fitting and option information in the blueprint is extracted, these constraints are checked by a C code module which uses the constraint information in the fitting knowledge base file to check the extracted options and arguments. Thus, when a change to the fitting is made which affects this constraint information, the constraint information in the fitting knowledge base file must be updated. Thereafter, when the fitting is parsed, the updated constraint information will be used.

The third type of constraint information is represented directly as C code. This flexibility is needed as there are certain forms of constraints among options which are not easily represented in general languages or may occur so infrequently as to be impractical to support in the general case. This type of constraint information is contained in an explicit C function, whose name is specified in the fitting knowledge base file. When changes to the fitting impact this information, the code relevant code must be modified, compiled, and re-linked.

Another type of knowledge encoded in a flexible fashion is the runtime models. This information indicates how much time each processing step will take as a function of parameters including: the option settings, the number of data records in the dataflow, and the computer being used. Fitting models to cover new fittings can be constructed in two ways. First, existing runtime models can be used as templates. In this case creating a runtime model for a new fitting coresponds to filling in the appropriate parameters in the model. Second, a new fitting model can be created from scratch (and would serve as a potential template for future fittings).

Discussion

The current prototype version of PIPE was completed in July 1991. It is implemented in CommonLISP and LISPView and runs on Sun workstations. It operates as described in this paper with the exception that it does not distinguish between columns accessed for different computations in a fitting (i.e. it only determines the set of columns used to compute all of the new or modified columns). For instance, suppose the runstat fitting uses column bx to create column rabx and also uses column by to create column raby. The current implementation will only be able to state that the the fitting uses columns bx and by to create columns rabx and raby. In contrast, the new implementation will be able to isolate bx as the column used to create column rabx, and by as the column used to create column raby. Also, the current prototype version operates on actual blueprint files but is not integrated with Master Plumber or MPTool, a menu driven interface for blueprint construction in Master Plumber.

Work is underway on the deliverable version of PIPE. This version is being implemented in C++, and is expected to be completed in May of 1992. The deliverable version of PIPE will use the more refined dependency representation described in this paper. This version will be integrated with Master Plumber and MPTool, and is intended to be delivered to and used by IGPP personnel at UCLA. This version of PIPE will also incorporate feedback upon the "look and feel" of the interface specified by IGPP personnel.

There are numerous related projects in providing intelligent assistance in scientific computing. The Kineticist's workbench project at MIT [Abelson et al. 1989] targets modelling and analysis of dynamic systems. The SINAPSE system [Kant et al. 1990] assists in construction of numerical models for data interpretation but is specific to seismic models represented as finite difference equations. The Reason system [Atwood et al. 1990] supports analysis of high energy physics data (and is a dataflow system). Finally, the Scientific Modeling Assistant project [Keller 1991] addresses support to facilitate development of scientific models.

Summary

This paper has described a system to assist in the development of scientific data preparation programs and discussed issues in design for maintainability. This issue of maintainability was particularly important because the processing modules (fittings) are constantly evolving due to changing scientists' needs. In order to maximize maintainability of the constraint knowledge base, information for each fitting is encapsulated in a fitting knowledge base file and as much as is practical, constraint information is represented in a general declarative fashion.

Acknowledgements

This work was performed by the Jet Propulsion laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[Abelson et al. 1989] H. Abelson, M. Eisenberg, M. Halfant, J. Katzenelson, E. Sacks, G. Sussman, J. Wisdom, and K. Yip, "Intelligence in Scientific Computing", Comm. ACM, 32(5):546-562, May 1989.

[Atwood et al. 1990] W. Atwood, R. Blankenbecler, P. F. Kunz, B. Mours & A. Weir, "The Reason Project", Stanford Linear Accelerator Technical Report #SLAC-PUB-5242, April 1990.

[Chien et al. 1992] S. Chien, R. K. Kandt, R. Doyle, J. Roden, T. King, and S. Joy, "PIPE: An Intelligent Scientific Data Preparation Assistant", Proceedings of the International Space Year Conference on Earth and Space Science Information Systems, Pasadena, CA, February 1992.

[Kant et al. 1990] E. Kant, F. Daube, W. MacGregor, J. Wald, "Synthesis of Mathematical Modeling Programs", Schlumberger Laboratory for Computer Science Technical Report Number TR-90-6, February 1990.

[Keller 1991] R. Keller, "Building the Scientific Modeling Assistant: An Interactive Environment for Specialized Software Design", Technical Report FIA-91-13, NASA Ames Research Center, Moffett, Field, CA, May 1991.

[King & Walker 1991] T. King and R. Walker, "The UCLA Data Flow System," Technical Report #3522, Institute of Geophysics and Planetary Physics, University of California at Los Angeles, CA 1991.

GATOR: Requirements Capturing of Telephony Features

Douglas D. Dankel II
ddd@cis.ufl.edu

Wayne Walker
ww0@cis.ufl.edu

Mark Schmalz
msz@mosquito.cis.ufl.edu

E301 CSE, C.I.S.
University of Florida
Gainesville, FL 32611
(904) 392-1387 (Office)
(904) 392-1220 (FAX)

1. Introduction

During the past twenty years the telecommunications industry has become increasingly dependent upon software-controlled switching systems. The software of these systems automates the billing of long distance calls, supports direct dialing of overseas calls, and provides features (e.g., call waiting, call forwarding) that many people consider essential components of everyday life. While telephony software has become both very large and complex in function and structure, the methods of software description have changed little over the past two decades. All existing characteristics and features as well as any modifications or additions to this software are described through natural language requirements specification documents.

These documents present a real dilemma to both the developers and customers. While these documents are essential to describe of the functionality of telephony features, the ambiguity and uncertainty inherent within natural language often leads to misinterpretations which can severely impact the resulting implementation of the functionality, the user acceptance of these features, and/or the development cycle.

We are developing a natural language-based, requirements gathering system called GATOR (for the GATherer Of Requirements) that assists in the development of more accurate and complete specifications of new telephony features. GATOR interacts with a feature designer who describes a new feature, set of features, or capability to be implemented. The system aids this individual in the specification process by asking for clarifications when potential ambiguities are present, by identifying potential conflicts with other existing features, and by presenting its understanding of the feature to the designer. Through user interaction with a model of the existing telephony feature set, GATOR constructs a formal representation of the new, "to be implemented" feature. Ultimately GATOR will produce a requirements document and will maintain an internal representation of this feature to aid in future design and specification.

This paper consists of three sections that describe (1) the structure of GATOR, (2) POND, GATOR's internal knowledge representation language, and (3) current research issues.

2. The Structure of GATOR

GATOR consists of three major components, illustrated in Figure 1:

1. **The User Interface** (consists of the Parser, Lexical & Grammatical Knowledge Base, Predicate Generator, and Response Generator) accepts natural language requirements descriptions and reports its understanding of these requirements to the user. Additionally, the User Interface answers user queries regarding the system's understanding of a feature and requests clarification of input which may be ambiguous or may contain recognizable errors.

2. **The Command Interpreter** (consists of the Interpreter) receives information and commands from the user interface, and issues queries and update instructions to the Knowledge Base/Data Base. This information specifies actions to be taken by the telephone switching circuits and software (e.g. "The Directory Number is always transmitted to the terminating office as a part of the Initial Address Message."), provides structural/organizational knowledge (e.g., "Calling Number Delivery Blocking (CNDB) is a CLASS feature."), or describes actions for displaying information (e.g., "Display a call with CNDB and Three-way Calling (3WC)."),

locating information within the representation (e.g., "What are the parts of a call?"), creating new knowledge that must be stored (e.g., "After the access code is entered, it is checked for validity."), or modifying existing knowledge (e.g., "The check for CNDB validity is made after the access code is verified as a valid code.").

3. **The Knowledge/Data Base** is a repository of information about the general structure of a call, existing features, and the new feature being defined. It consists the three levels, described in the next section.

3. Knowledge/Data Base

The Knowledge/Data Base contains specific knowledge of the components of a call and all existing features. It was built using POND [DANK92] (the Pantological Organization of New Delineations), a knowledge representation structure based on the family of KL-ONE languages [BRAC85, BRAC89, WOOD90]. While most of the KL-ONE languages divide knowledge into two partitions, called the Terminological Box or TBox and the Assertional Box or ABox, POND consists of three distinct knowledge levels as shown in Figure 2:

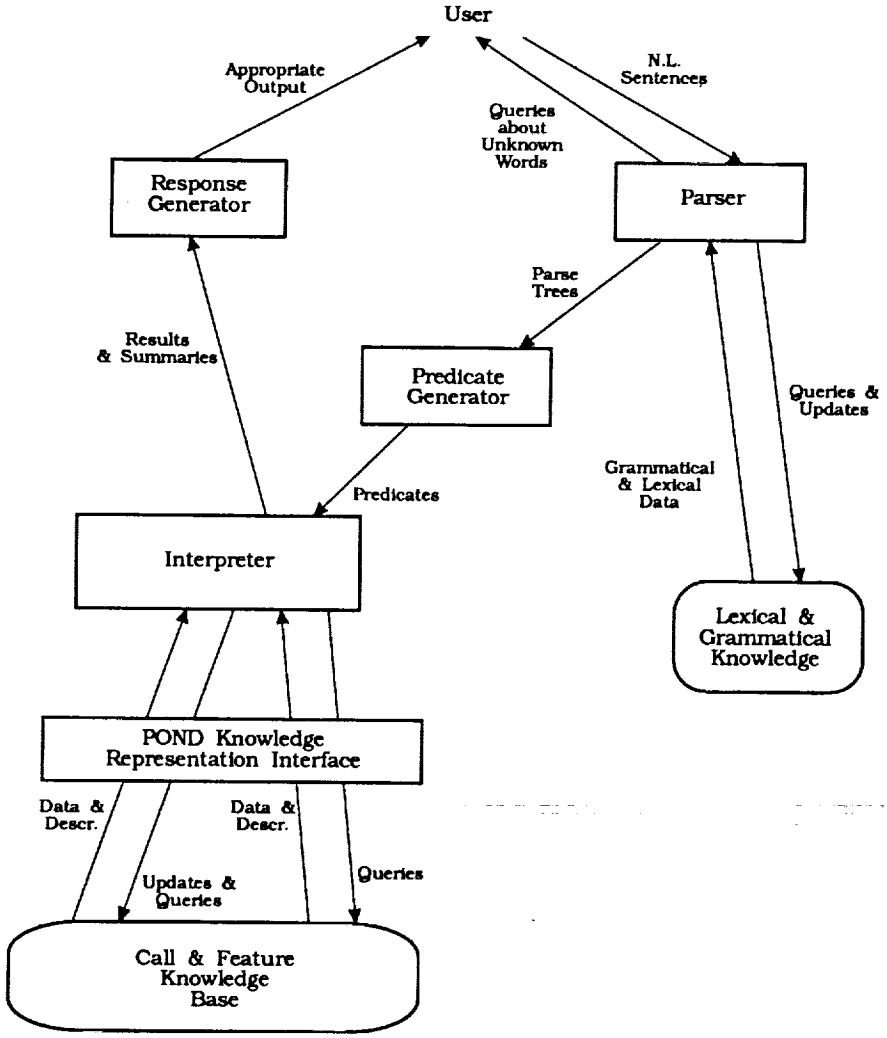


Figure 1. Internal System View

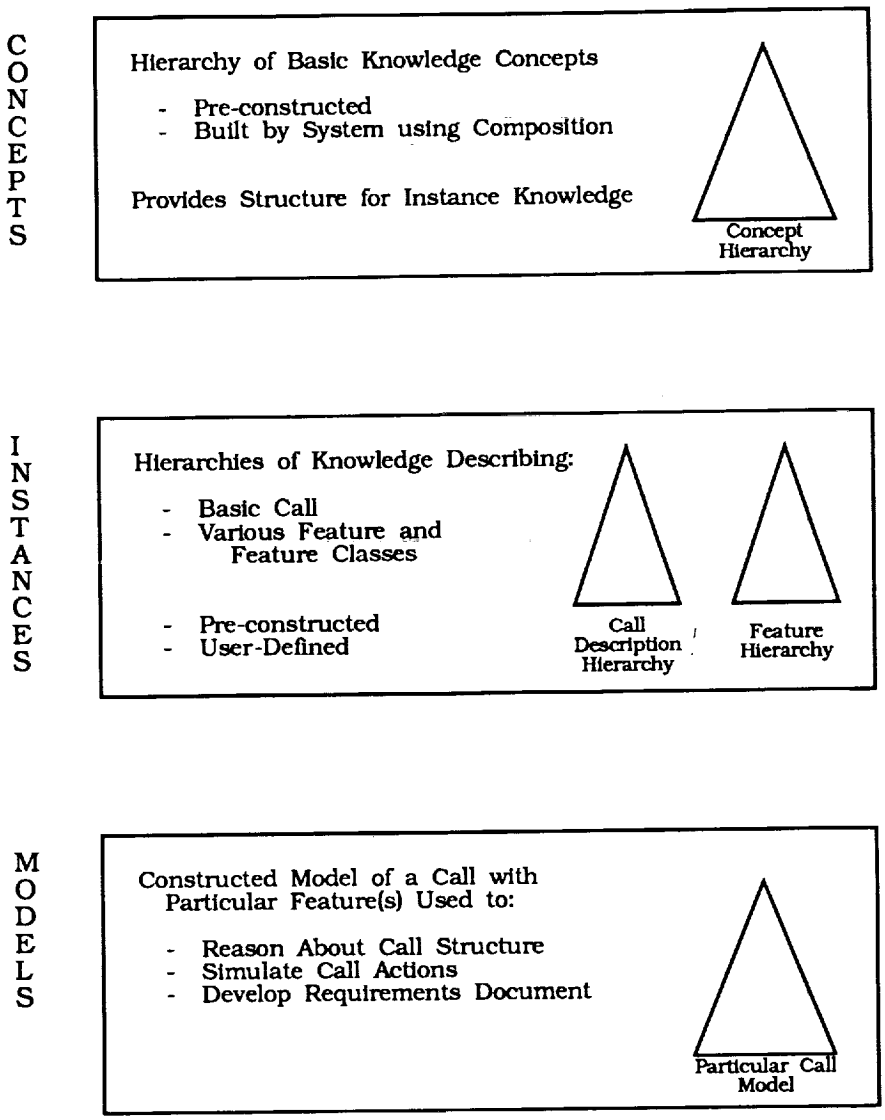


Figure 2. A Conceptual Diagram of the Knowledge Levels within the Call & Feature Knowledge Base

1. **Concepts.** High-level conceptual knowledge used to structure all of the knowledge within the knowledge base.

2. **Instances.** Specific descriptions of the components of a call, existing features, and the dynamic specification of the feature under definition.

3. **Models.** A constructed model of a particular call containing specific features.

A short description of each of these components follows.

3.1. Concepts

Knowledge on the Concept Level provides a structure for the knowledge on the Instance and Model Levels. Conceptual knowledge includes definitions of:

1. The concepts that represent telephone call and feature components. Each concept contains several slots (i.e., `:features`) that define the type and number of permitted values. Concepts can additionally include references (i.e., `:ako`) to other concepts on which they are based and applicable constraints (i.e., `:annotation`).

2. Special slots, or attributes, of the concepts, which define a set of restricted values or define relationships between concepts. For example, the `category` slot defines a restricted set of allowable slot values, while the `children` and

parent slots define relationships between slots.

3. Temporal relations [ALLE85] required for specifying temporal ordering within instances and models.

3.2. Instances

The Concept Level defines knowledge fundamental to instances on the Instance Level. A particular concept associates with each instance providing a structure and certain internal values for the instance. Instance knowledge includes descriptions of call and feature components. For example, a *call* initially decomposes into the logical components (instances) of *go-off-hook*, *make-call*, and *disconnect-call*. Each of these components is, in turn, further decomposed on the Instance Level. The temporal relationships associated with each instance define the instance's location to the other instances.

Instance Level knowledge also includes descriptions of the various telephony features. Each feature, such as Three-Way Calling (3WC) and Calling Number Delivery Blocking (CNDB), decomposes into a structure similar to the decomposition of a *call* shown in Figure 3. These decompositions detail the individual operator actions that enable each feature, resultant system actions, and temporal relationships between feature components and *call* components.

Besides modeling individual features, the feature descriptions contain restrictions and special interactions between features. For example, since the features of 3WC and CNDB are compatible but interact, the interaction must be specified. See Figure 4.

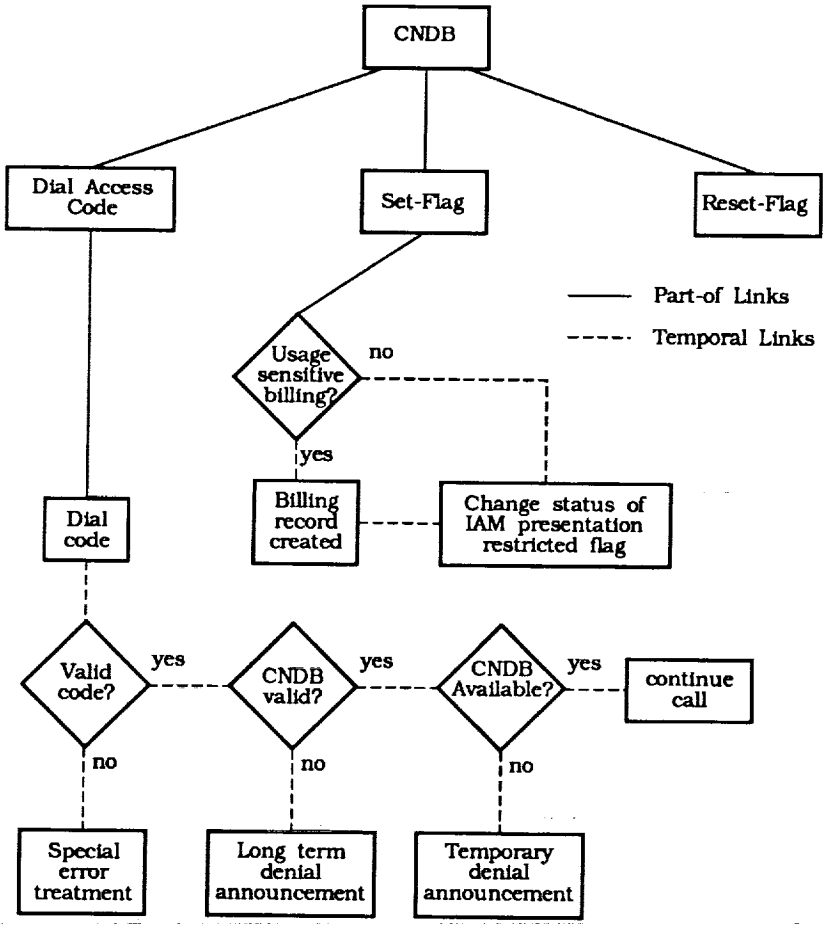


Figure 3. Decomposition of *CNDB*

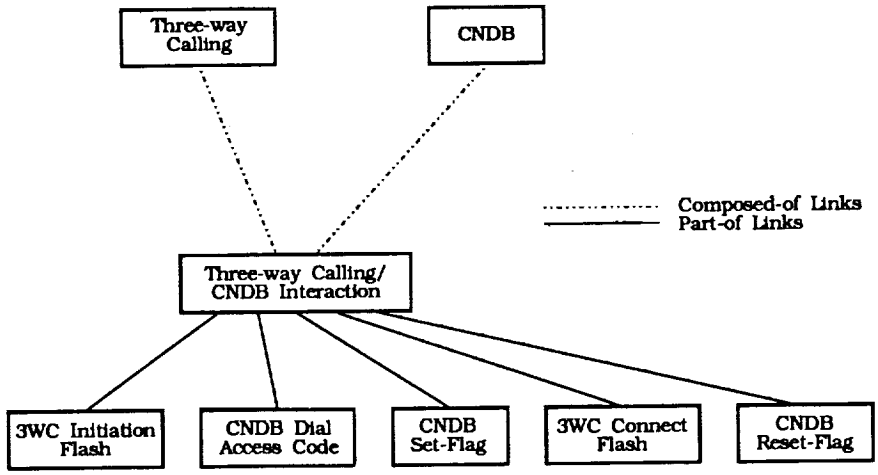


Figure 4. Feature Interaction with the Feature Hierarchy

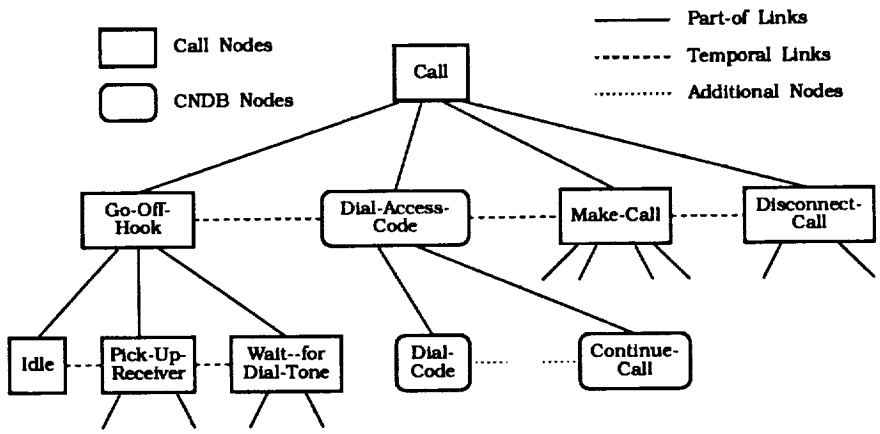


Figure 5. Model Representation of a *Call* with *CNDB*

3.3. Models

The Model Level facilitates the building of a description of a particular telephone call exhibiting specific features. While the knowledge on the Concept Level changes very little (due to the operation of Composition [DANK92]) and the primary goal of GATOR is for the user to build Instance Level knowledge of some new feature, the Model Level is significantly more dynamic.

While operating GATOR, the user can request a description of a telephone call which exhibits a particular feature or set of features. The system examines its Instance Level knowledge and retrieves the appropriate instances representing a general call and the set of features of interest to the user. These instances combine to develop a model of the requested call type as shown in Figure 5. The instance ordering depends upon any explicit or implicit interactions and dependencies that exist between features and the feature specification order.

Upon completion, models are presented to the user. The display of a model allows the user to verify that knowledge represented on the Instance Level is correct and complete. Errors detected generally result from incomplete or incorrect specifications on the Instance Level. Each must be identified and corrected by the user. After locating and correcting an error, the user can verify that appropriate corrections were made by creating another model and examining its revised structure.

4. Status and Plans

Our current research in automated requirements gathering includes:

1. **Improvements and extensions to POND.** POND, as originally constructed, provides a rich environment for specifying set/member and part/sub-part relations. While the system currently includes the ability to specify temporal information, it does not provide a unified temporal reasoning component or subsumption, each of which require further definition and incorporation.

2. **Expansion of the knowledge base to include additional feature knowledge.** Currently, the feature knowledge base consists of a limited set of telephony features. This knowledge base needs to be significantly expanded to provide an adequate environment for developing new features, specifying feature interactions, testing model building, and expanding the natural language interaction.

3. **Development of a Specification Document Generator.** Once feature knowledge has been captured within GATOR's knowledge base, it must be made more accessible to developers, implementors, and customers. An output generation system is currently under design with which the user will be able to produce a feature requirements specification document.

4. **Extension of the Natural Language Capabilities.** The current system is limited in the range of input which is can process. We are expanding the syntactic and semantic capability of the system to more closely model the range of language used by designers when they describe a feature's structure.

While our research has concentrated on telephony, our approach is applicable to a wide range of domains. An initial examination of telephony features has shown that GATOR can capture 80 to 90 percent of the functional requirements of a feature contained in a typical specification document. We expect that the use of such an automated tool, in this and other domains, will significantly reduce ambiguities and uncertainties within specification documents, thereby decreasing development time and expense.

5. References

[ALLE85] Allen, J., *Maintaining Knowledge about Temporal Intervals*, in **Reading in Knowledge Representation**, edited by R. J. Brachman and H. J.. Levesque, Morgan Kaufman, Los Altos, CA, pp. 509 - 521, 1985.

[BRAC85] Brachman, R. J. and J. G. Schmolze, *An Overview of the KL-ONE Knowledge Representation System*, **Cognitive Science**, Vol. 9, No. 2, pp. 171 - 216, 1985.

[BRAC89] Brachman, R. J., A. Borgida, D. L. McGuinness, and L. Alperin Resnick, *The CLASSIC Knowledge Representation System, or, KL-ONE: The Next Generation*, **Workshop on Formal Aspects of Semantic Networks**, Santa Catalina Island, CA, 1989.

[DANK92] Dankel, D. D., W. Walker, and M. Schmalz, *POND: A Knowledge Representation Language which Facilitates Requirements Capturing*, Working Paper submitted to the **12th International Avignon Conference**, 1992.

[WOOD90] Woods, W. and J. Schmolze, *The KL-ONE Family*, TR-20-90, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1990.

Modeling Software Systems by Domains

Richard D'Ippolito and Kenneth Lee
Software Engineering Institute
Carnegie Mellon University

The Software Architectures Engineering (SAE) Project at the Software Engineering Institute (SEI) has developed engineering modeling techniques that both reduce the complexity of software for domain-specific computer systems and result in systems that are easier to build and maintain. These techniques allow maximum freedom for system developers to apply their domain expertise to software.

We have applied these techniques to several types of applications, including training simulators operating in real time, engineering simulators operating in non-real time, and real-time embedded computer systems. Our modeling techniques result in software that mirrors both the complexity of the application and the domain knowledge requirements. We submit that the proper measure of software complexity reflects neither the number of software component units nor the code count, but the locus of and amount of domain knowledge. As a result of using these techniques, domain knowledge is isolated by fields of engineering expertise and removed from the concern of the software engineer. In this paper, we will describe kinds of domain expertise, describe engineering by domains, and provide relevant examples of software developed for simulator applications using the techniques.

Separation of Concerns by Domain Expertise

We classify computer system developers by expertise and role using three categories: systems analyst, domain engineer, and software engineer. Systems analysts are responsible for defining the policy, strategy, and use of the application to be developed, e.g., the concept of operations, and the training requirements. Domain engineers are the modelers responsible for determining which real-world entities need to be modeled to satisfy the policy, strategy, and use defined by the systems analysts. Domain engineers determine if and how the entities selected to be modeled can be specified within the constraints imposed by the software engineers. Finally, they express the models in the language natural to their domain. Software engineers are responsible for defining a consistent software structure into which the domain expertise will go, and providing translations from the domain-specific natural languages into executable software.

It is not generally possible to reduce the amount of domain knowledge required to either develop or enhance a software-dependent system. To borrow a phrase from Albert Einstein, our system models should be as simple as necessary, but no simpler. If we can separate the design of the models from the design of the software, we can separate the tasks of the domain engineer from the tasks of the software engineer. This would allow the software engineer to make simplifications in the software packaging and execution structures which would not affect the way the domain engineer expresses the models. It would also allow the domain engineer the freedom to design model algorithms without requiring specialized software knowledge. In effect, each engineer is relieved of the burden of becoming an expert in other domains of expertise.

We have found that this separation of concerns by domain expertise is what enables us to simplify the overall design process and gain a more enhanceable (maintainable) computer system.

Engineering by Domain

In our vocabulary, a *domain* is a specific field of engineering expertise. Engineering expertise is classified by families of models and related sets of practices for applying the models, not by the problems to which the expertise is applied. Common classifications of engineering domains are: electrical, civil, nuclear, mechanical, chemical, and (the as yet undefined field of) software engineering. An *application area* consists of related problems that can be described using models from a variety of domains. Examples of application areas are command and control systems, factory automation

This work is sponsored by the U.S. Department of Defense. The SAE project members are Richard D'Ippolito, Kenneth Lee, Charles Plinta, and Jeffrey Stewart.

systems, embedded systems, and simulator systems[1]. Thus, a flight simulator application requires domain expertise in aeronautical engineering, electrical engineering, mechanical engineering, and so on.

Models are reusable, adaptable, engineering assets because they are patterns expressed in their most general form and are scalable, usually through *templates*. A good example of a templated model is a dress pattern, where all of the cut-lines are given by dress size.

We classify models using two major types, which we call *product models* and *practice models*[2]. The product model, when scaled, results in a component of the delivered product. The dress pattern is an example of a product model, as is the set of engineering drawings for an I-beam or a DC motor. Clearly, the dress pattern is no good without the practice know-how of fabric and thread selection, cutting, stitching, hemming, pleating, and all of the other activities needed to produce the final product. As a commercial venture, dress-making would require in addition to the product models the assembly-line models, materials-handling models, business and economic models, and so on. All of these models are what we call the practice models, because they define the established body of practice around the product models. Interestingly, the more mature an engineering discipline, the more the product and practice models will be public. In a mature discipline, the business enterprise seeks value added through system composition (model application), not model creation or refinement, which are seen as adjunct activities to be undertaken only when necessary to complete an application.

In the construction industry (civil engineering and architecture), for example, all engineering firms have access to the same materials, material costs, implementation practice (labor), and labor costs. In these cases, the firms compete on system composition, where success is meeting the customer's needs with a timely and economical design. Electrical engineers do not manufacture their own wire, integrated circuits, resistors, and other electrical and mechanical components, but compete on the basis of using these components efficiently to satisfy a need. The information on the components themselves is found in *engineering databooks* (usually manufacturer's publications), and *engineering handbooks* which are compendia of the practice knowledge. Both require an experienced practitioner with an in-depth education to interpret, however, as one cannot learn and practice an engineering discipline solely from the handbooks. With that education, however, the use of the handbooks will go a long way toward guaranteeing a successful routine (precedented) design. The use of the handbooks are not intended to support innovative design.

SAE has been very successful in applying models across various software application areas because our models have captured patterns of structure and behavior at the domain level. The *Object-Connection-Update (OCU)* model[3] is a good example of a building block that allows the domain engineer to capture the patterns of structure and behavior of the real-world subsystems being modeled[4]. Originally created for flight simulators, the OCU was immediately applied to the design of the seeker subsystem of an anti-tank missile and is now being used in the design of subsystems for engineering simulators. What made these applications of the model possible was the capturing of the basic pattern of subsystem operation into a few standardized architectural elements[5] (models), each responsible for a particular subsystem task. Complexity is reduced because any subsystem can (and must) be expressed using only these basic elements, thus constraining the choice of solution structures available for consideration. Systems analysts, domain engineers, and software engineers

1. As an example, consider the domain of a rope where force is transmitted through tension in a flexible member (try using a rope under compression to push an object). Mechanical engineers have no problem applying the same rope design models, i.e., the domain expertise, to suspension bridges, elevators, cranes, and fishing rods, yet the application areas will seem quite unrelated to those not proficient in the domain.
2. We have deliberately avoided the overloaded term *process*, preferring to reserve it for its traditional engineering reference to a controlled activity within a plant or machine. We use *practice* to refer to those engineering activities that support product development.
3. The seminal report on the OCU is CMU/SEI-88-TR-30, *An OOD Paradigm for Flight Simulators, 2nd Edition*. This report, however, is dated relative to current SAE experience and is being updated. We are, also, in the process of writing a series of white papers that will fully describe the OCU and the engineering of software-dependent systems.
4. In our terms, the total application is composed of *subsystems* so that those who wish may apply the term *system* to the whole.

are able to make use of the OCU as the basis for their separation of concerns; the OCU is the framework that ensures all activities will work together.

OCU Subsystem Examples

The OCU, produced by the software engineers, guides the systems analysts and domain engineers by providing the fundamental pattern of analysis and the structure for model capture. The systems analysts, with the foreknowledge that the ultimate software implementation will be subsystems captured by the OCU, will be guided to view the application as a collection of subsystems. The domain engineers, with the same foreknowledge, will be guided to compose models as collections of subsystems, each composed of objects organized by a controller. We will show in the following examples, taken from a simulator application, how the OCU provides this guidance.

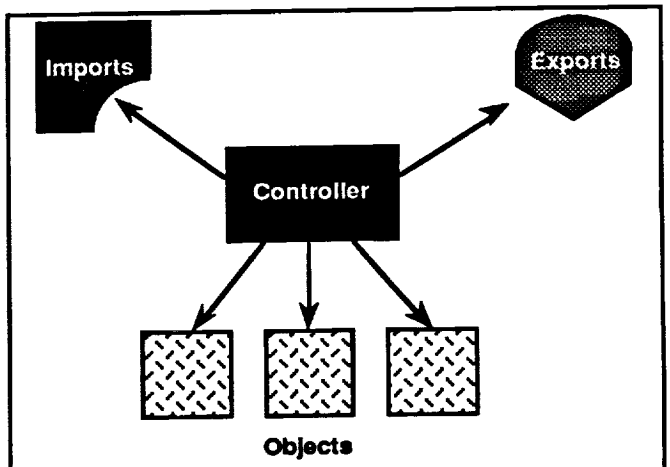


Figure 1: OCU Subsystem Diagram

Before we describe how the OCU provides this guidance, we will provide more detail about the OCU

5. The basic elements are controllers, objects, import areas, export areas, surrogates, and device handlers. Controllers are the loci of subsystem connection and operation information; objects provide the subsystem services; import areas provide the subsystem with a view to the external world; export areas provide a window into the subsystem state for the external world; surrogates translate information from external formats to internal formats and back; and device handlers handle external-world communications. All instances of each of these elements are of the same form (implementation structure).

Subsystem Form

Subsystem Name: _____

Description: _____

Overview of Requirements:

Objects:

Imports:

Name	Type	Source

Exports:

Name	Type	Destination

Update Algorithm:

Controller Template

```
package <subsystem_name>_Controller is

      -- every subsystem controller has an update procedure
      -- called by the executive
      procedure Update;

end <subsystem_name>_Controller;



with SEU;  -- global types
with <subsystem_name>_Types; -- the 'local' types
with <subsystem_name>_Imports;
with <subsystem_name>_Exports;

-- all objects that are part of this subsystem
with <Object_1>_Manager;
with <Object_2>_Manager;
with <Object_3>_Manager;
with <Object_4>_Manager;
with <Object_5>_Manager;

package body <subsystem_name>_Controller is
      -- local variables declared here
      type <type1>;
      type <type2>;

      procedure Update is
        begin
            -- controller update algorithm goes here
        end Update;

end <subsystem_name>_Controller;
```

Figure 2: Subsystem Specification Form and Controller Template

itself. We have found that the general patterns of operation of subsystems in any domain can be captured in a universal structure. These patterns involve separation of mission from operation, localization of state, activation and control of subsystems, and transfer of information. Separation of mission from operation is derived from a principle that is fundamental to all human and machine behavior: the mechanism of making decisions should be separate from the mechanisms used to carry out the decisions. Localization of state is derived from the fundamental software engineering principle of information hiding. In the OCU (Figure 1), the controller is the locus of decision making, and the objects provide the service mechanisms and the localization of state.

We knew that we could reduce the software complexity by repeated use of a small number of elements, a standard method of information transfer, and a standard method of control. We also knew that a maintainable system required closely related services be isolated from other, unrelated, services. In software engineering terms, this means coupling between unrelated entities is minimized, cohesion between related entities is maximized, and maintainability is enhanced by repeated use of the same patterns. In the OCU, isolation and information transfer is provided by the import and export areas. Cohesion among the objects in a subsystem is enforced by having the controller be the sole entity that implements connections to objects. We have found this set of elements: objects, controllers, export areas, and import areas, to be sufficient for describing any real-world subsystem.

We, as software engineers, have implemented the elements of the OCU in Ada. We have captured the patterns with a subsystem specification form and a set of element code templates.

The OCU is applied with the aid of the subsystem specification form and the element code templates, subsets of which are shown in Figure 2 (only the controller template is shown). The subsystem form provides a standard way for the systems analyst and domain engineers to record the specifications of subsystems in terms of the known compositional elements of subsystems, as shown in Figure 3. The subsystem templates provide a standard way for the

Sonar Subsystem Form

Subsystem Name: *Sonar*

Description:
The sonar subsystem is used to locate mine-like objects. Its transmit power level and pulse repetition rate are controlled by the console operator. The received signals are sent to the console.

Overview of Requirements:

References:	SW5TO-EO-MMO-020
	pp. 3-5
	pp. 7- all
	FO-8
	FO-12
	Telemetry Data Format
	MNV-Engineering Worksheet
	Schematic Slide

Objects:
- Sonar Soundhead
- Sonar Tilt Potentiometer
- Flow Control Servo_Valve
- Rotary Actuator

Imports:

Name	Type	Source
Rate_Cmd	Volts	Electronics Unit
Xmit_Level_Cmd	Xmit_Level	Electronics Unit
Slew_Rate_Limit_Cmd	Slew_Rate_Limit	Electronics Unit
Range_Reset_Cmd	Range_Reset	Electronics Unit
Sonar_Received_Signal	Sonar_Signal	Environment
Pulse_Repetition_Rate_Cmd	Pulse_Repetition_Rate	Electronics Unit
Hydraulic_Pressure_Available	Hydraulic_Pressure	Hydraulic System

Exports:

Name	Type	Destination
Sonar_Tilt_Potentiometer_Voltage	Volts	Electronics Unit
Composite Video	Sonar_Video_Signal	Electronics Unit
Sonar Transmitted Signal	Sonar_Signal	Environment

Sonar Controller Code

```
package Sonar_Controller is

      -- every subsystem controller has an update procedure
      -- called by the executive
      procedure Update;

end  Sonar_Controller;


with SEU;  -- global types
with Sonar_Types; --  the 'local' types
with Sonar_Imports;
with Sonar_Exports;

-- all objects that are part of this subsystem
with Flow_Control_Servo_Valve_Manager;
with Rotary_Actuator_ Manager;
with Sonar_Soundhead_ Manager;
with Sonar_Tilt_Potentiometer_ Manager;

package body Sonar_Controller is

    procedure Update is
     begin
        Flow_Control_Servo_Valve_Manager.Update(
                Sonar_Imports.Rate_Command,
                Sonar_Imports.Hydraulic_Pressure_Available,
                Sonar_Exports.Controlled_Pressure);

        Rotary_Actuator_Manger.Update(
                Sonar_Exports.Controlled_Pressure,
                Sonar_Exports.Controlled_Torque);
```

Figure 3: Completed Subsystem Specification Form and Controller Template (truncated)

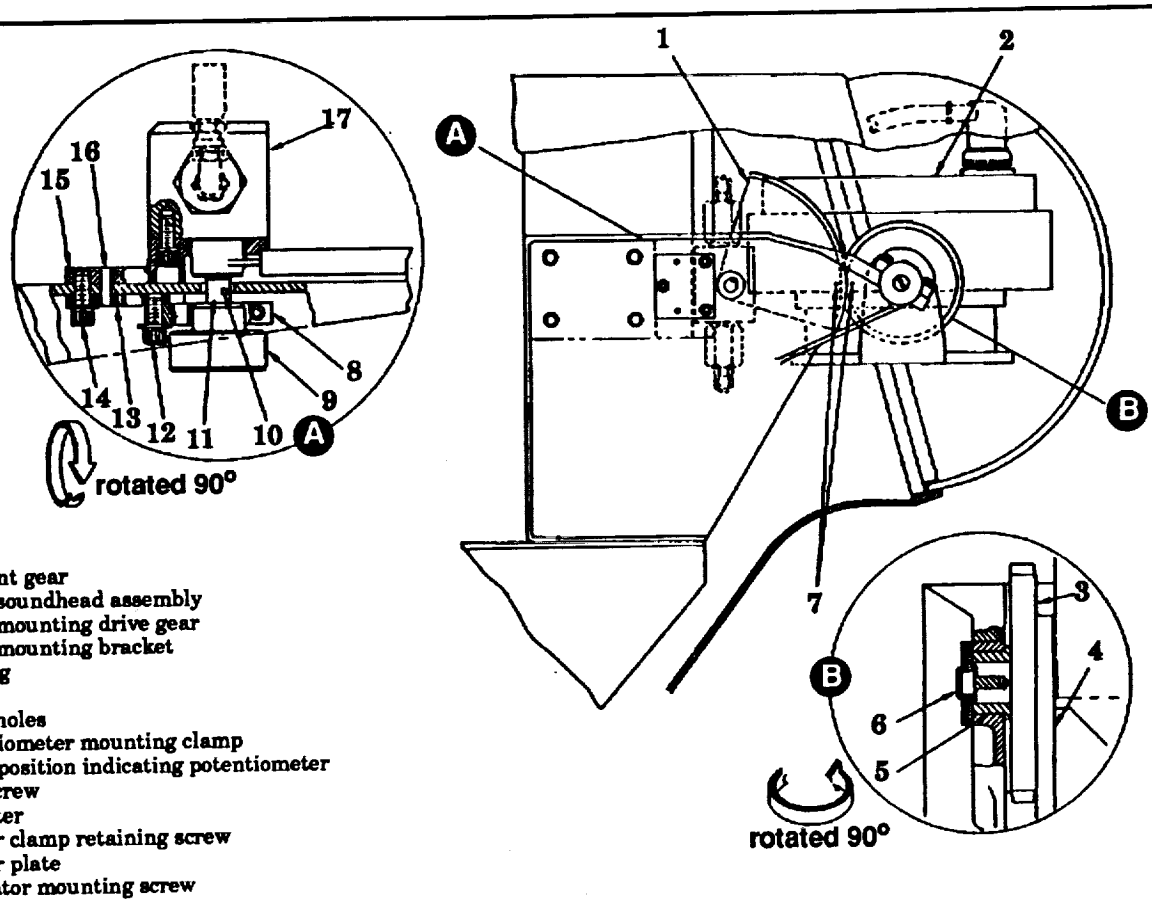
software engineer to map the design of the models, captured on the forms, directly into an Ada implementation of the elements, also shown in Figure 3.

We can now describe how the OCU provides guidance to systems analysts and domain engineers. The systems analyst, in consultation with the customers and users, analyzes the application to identify subsystems consistent with the concept of operation and patterns of use of the application. Each of the subsystems is assigned a specification form and passed to the appropriate domain engineer for completion. In addition to the identification of subsystems, the systems analyst will provide the domain engineer with a mapping of the training requirements expressed in terms of model fidelity, operational modes, and malfunctions.

Figure 4 shows a sonar subsystem schematic from a Navy remote-controlled, minehunting, undersea vehicle. This diagram was constructed by sonar engineers and represents the real-world sonar subsystem. The schematic captures the knowledge needed by the domain engineer to model how the sonar subsystem is constructed. For the construction of a complete simulator, the systems analyst will gather representative schematics and provide them, with the specification forms, to domain engineers.

A domain engineer receives a partially completed form and some subsystem schematics from the systems analyst. The domain engineer then models the real-world subsystem to match the fidelity requirements expressed on the form. Each element of the model is mapped to an element of the OCU, the element models are parameterized to realize the specified operational modes and malfunctions, and the parameterized models are captured in a language natural to the domain engineer. The domain engineer completes the specification form by recording the mapping and forwarding the form, containing the natural language description of the parameterized models, to the software engineer.

Figure 5 shows a representation of the sonar subsystem as modeled by the domain engineer. The objects remaining are those sufficient to simulate the subsystem to match the fidelity requirements, modes, and malfunctions. Some connections to other



Legend
1. Segment gear
2. Sonar soundhead assembly
3. Sonar mounting drive gear
4. Sonar mounting bracket
5. Bearing
6. Screw
7. Index holes
8. Potentiometer mounting clamp
9. Sonar position indicating potentiometer
10. Set screw
11. Adapter
12. Sonar clamp retaining screw
13. Shear plate
14. Actuator mounting screw
15. Actuator mount
16. Alignment spring pin
17. Rotary actuator

Figure 4: Sonar Subsystem Schematic

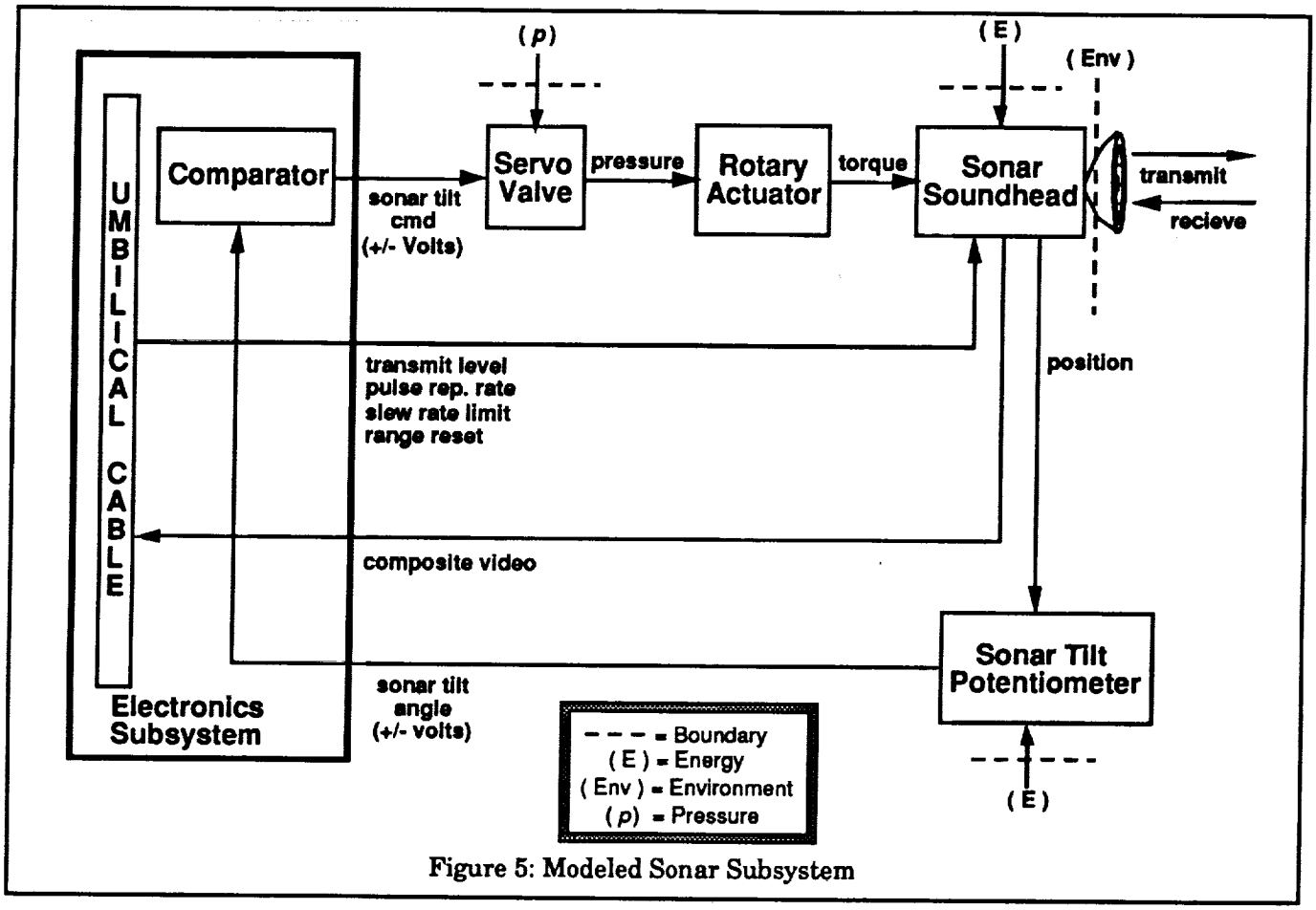


Figure 5: Modeled Sonar Subsystem

subsystems on the undersea vehicle are shown as well. Figure 6 shows an OCU diagram for the modeled sonar subsystem.

Conclusions

Using a fixed set of templates means that the interface mechanism between elements is known ahead of and independent of model design. All subsystems look (structurally) alike, and each subsystem can be made to lie within a single domain, with communication between subsystems also being handled by common structures. This means that the software engineer can proceed with executive and test harness design. It also means that the model specifiers can work independently in their own domains, knowing that their models will fit into the completed system.

Thus, a completed simulator application will consist of as many instances of the OCU subsystem model as required by the use and fidelity requirements. Space limitations prevent us from describing the additional elements used to compose the simulator executives, but the same techniques and the OCU are used there as well.

We conclude that composition by domain-specific subsystems allows maximum freedom for the systems analysts, domain engineers, and software engineers to apply their expertise, and that having common software structures results in software applications that are more easily understood and enhanced, i.e., systems which have reduced complexity.

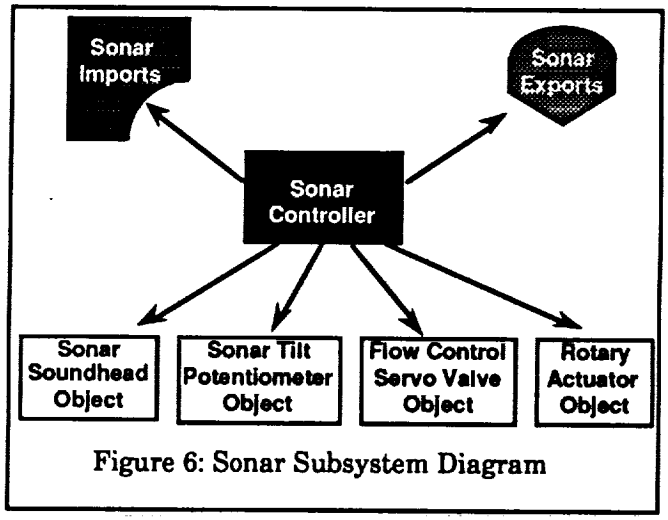


Figure 6: Sonar Subsystem Diagram

Approximation, Abstraction and Decomposition in Search and Optimization

Thomas Ellman
Department of Computer Science
Rutgers University
ellman@cs.rutgers.edu

1. Synthesis of Search Control Heuristics

One portion of my research has focused on automatic synthesis of search control heuristics for constraint satisfaction problems (CSPs). I have developed techniques for automatically synthesizing two types of heuristics for CSPs: Filtering functions are used to remove portions of a search space from consideration. Evaluation functions are used to order the remaining choices. My techniques operate by first constructing exactly correct filters and evaluators. These operate by exhaustively searching an entire CSP problem space. Abstracting and decomposing transformations are then applied in order to make the filters and evaluators easier to compute. An abstracting transformation replaces the original CSP problem space with a smaller abstraction space. A decomposing transformation splits a single CSP problem space into two or more subspaces, ignoring any interactions between them. Both types of transformation potentially introduce errors into the initially exact filters and evaluators. The transformations thus implement a tradeoff between the cost of using filters and evaluators, and the accuracy of the heuristic advice they provide. I have shown these techniques to be capable of synthesizing useful heuristics in domains such as floor-planning and job-scheduling, among others. (See [Ellman, 1992].)

2. Synthesis of Hierarchic Problem Solving Algorithms

Another portion of my research is focused on automatic synthesis of hierarchic algorithms for solving constraint satisfaction problems (CSPs). I have developed a technique for constructing hierarchic problem solvers based on numeric interval algebra. My system takes as inputs a candidate solution space S and a constraint C on candidate solutions. The solution space S is assumed to be a cartesian product R^n where R is a set of integers. The constraint C is assumed to be represented in terms of arithmetic, relational and boolean operations. From these inputs the system constructs an abstract solution space S_a as a cartesian product R_a^n where R_a is a set of disjoint intervals that covers R. The system also constructs an abstract constraint C_a on abstract solutions. The abstract constraint C_a is obtained from the original constraint C by replacing ordinary arithmetic operations with interval algebra operations and replacing boolean operations with boolean set operations. The abstract space S_a and abstract constraint C_a are then used to build a hierarchic problem solver that operates in two stages. The first stage finds an abstract solution in the space S_a of intervals. The second stage refines the abstract solution into a concrete solution in the original search space S. I have shown this approach to be capable of synthesizing efficient problem solvers in domains such as floor-planning and job-scheduling, among others. (See [Ellman, 1992].)

3. Decomposition in Design Optimization

Another portion of my research is focused on automatic decomposition of design optimization problems. We are using the design of racing yacht hulls as a testbed domain for this research. Decomposition is especially important in the design of complex physical shapes such as yacht hulls. Exhaustive optimization is impossible because hull shapes are specified by a large number of parameters. Decomposition diminishes optimization costs by partitioning the shape parameters into non-interacting or weakly-interacting sets. We have developed a combination of empirical and knowledge-based techniques for finding useful decompositions. The knowledge-based method examines a declarative description of the function to be optimized in order to identify parameters that potentially interact with each other. The empirical method runs computational experiments in order to determine which potential interactions actually do occur in practice. We expect this approach to find decompositions that will result in faster optimization, with a minimal sacrifice in the quality of the resulting design. Implementation and testing of this approach are currently in progress. (I am pursuing this research in collaboration with Mark Schwabacher.) (See [Ellman *et al.*, 1992].)

4. Model Selection in Design Optimization

Another portion of my research is focused on intelligent model selection in design optimization. The model selection problem results from the difficulty of using exact models to analyze the performance of candidate designs. For example, in the domain of racing yacht design, an exact analysis of a yacht's performance would require a computationally expensive solution of the Navier-Stokes equations. Approximate models are therefore needed in order diminish the costs of analyzing and evaluating candidate designs. In many situations, more than one approximate model is available. For example, in the yacht design domain, the induced resistance of a yacht can be predicted by solving La Place's equation - an approximation of Navier-Stokes - or by using a simple algebraic formula. The two approximations differ widely in both the costs of computation and the accuracy of the results. Intelligent model selection techniques are therefore needed to determine which approximation is appropriate during a given phase of the design process.

We have attacked the model selection problem in the context of hillclimbing optimization. We have developed a technique which we call "gradient magnitude based model selection". This technique is based on the observation that a highly approximate model will often suffice when climbing a steep slope, because the correct direction of change is easy to determine. On the other hand, a more accurate model will often be required when climbing a gradual incline, because the correct direction of change is harder to determine. Our technique operates by comparing the estimated error of an approximation to the magnitude of the local gradient of the function to be optimized. An approximation is considered acceptable as long as the gradient is large enough, or the error is small enough, so that each proposed hillclimbing step is guaranteed to improve the value of the goal function. Implementation and testing of this approach are currently in progress. I am pursuing this research in collaboration with John Keane. (See [Ellman *et al.*, 1992].)

References

T. Ellman, J. Keane, and M. Schwabacher. The rutgers cap project design associate. Technical Report CAP-TR-6, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1992.

T. Ellman. Idealization-based methods for constraint satisfaction problems. Working Notes of the AAAI Workshop on Approximation and Abstraction of Computational Theories (Forthcoming), July 1992.

Meta-Tools for Software Development and Knowledge Acquisition

Henrik Eriksson* Mark A. Musen

Medical Computer Science Group
Knowledge Systems Laboratory
Stanford University School of Medicine
Stanford, CA 94305-5479

"Man is a tool-using animal.... Without tools he is nothing, with tools he is all."
Thomas Carlyle (1795–1881)

Abstract

The effectiveness of tools that provide support for software development is highly dependent on the match between the tools and their task. Knowledge-acquisition (KA) tools constitute a class of development tools targeted at knowledge-based systems. Generally, KA tools that are custom-tailored for particular application domains are more effective than are general KA tools that cover a large class of domains. The high cost of custom-tailoring KA tools manually has encouraged researchers to develop *meta-tools* for KA tools. Current research issues in meta-tools for knowledge acquisition are the specification styles, or *meta-views*, for target KA tools used, and the relationships between the specification entered in the meta-tool and other specifications for the target program under development. We examine different types of meta-views and meta-tools. Our current project is to provide meta-tools that produce KA tools from multiple specification sources—for instance, from a task analysis of the target application.

Introduction

Knowledge-acquisition (KA) tools are programs that help developers to elicit and structure domain knowledge for use in application programs (e.g., in expert systems). Typically, KA tools allow nonprogrammers who are specialists in some domain area to enter structures relevant for the application program without the aid of an intermediary who is proficient in programming. Thus, KA tools are, in a way, code-generating software-engineering tools for a restricted type of software and for a particular group of users. To increase the usability of KA tools, researchers in knowledge acquisition have experimented with specializing the tools in various ways. For instance, KA tools have been specialized to knowledge-acquisition methods, problem tasks, domains, and even applications. In most cases, specialized KA tools are reported to be more effective than general ones, because the users are nonprogrammers familiar with the domain terminology. In addition to those that involve the elicitation of knowledge from experts, there are approaches to KA tool support that rely on knowledge acquisition from texts, and there also are methods that incorporate machine learning from example solutions.

Custom-tailoring KA tools can be a laborious task. When the benefit of domain-specific KA tools is compared to the effort of developing them, the tool-development cost is often unacceptable for small projects. Also, development of domain-specific tools can in itself be a software-engineering problem. These problems have been addressed with supportive tools and, to a certain extent, with tool-development methodologies. Just as code-generator writing systems can be used to produce code-generating tools, *meta-tools* for knowledge acquisition can help developers to implement domain-specific KA tools. Several meta-tools that generate target KA tools automatically from specifications provided by the developers have been implemented by researchers in knowledge acquisition. Although KA tools are generally intended for nonprogrammers, variants of such tools can be used by programmers to increase software quality and programmer productivity. A meta-tool can be used to create the tool required by programmers.

An important aspect of a meta-tool is the specification strategy, or *meta-view*, for target tools that the meta-tool provides to the developers. The meta-view comprises the conceptual model of the target tool that the meta-tool supports, as well as the specification language for target tools. Depending on the view of target tools, several types of meta-views are possible. Domain-specific tools for software development are desirable in many situations. Meta-tools, however, preferably should be domain-independent so that they can produce domain-oriented tools for a broad area of applications.

Much of the work in meta-tool support for knowledge acquisition is relevant for software engineering, especially approaches to domain-specific development tools. If the design and implementation of such domain-oriented software-engineering tools are

*On leave from the Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden

laborious tasks, meta-level tools are certainly required. In this paper, we discuss alternative meta-views and describe their implementation in different meta-tools.

Background

Knowledge engineering and software engineering are partially overlapping disciplines. Moreover, tools for knowledge engineering and computer-aided software engineering (CASE) tools have gone through similar development stages, in the sense that increasingly specialized tools have been considered. The first-generation AI development tools were general and were essentially programming languages with integrated development environments. Examples of such tools are EMYCIN, KEE, ART, and S1. Only skilled programmers and knowledge engineers could use these tools, so the tools were inaccessible to domain specialists who had not had extensive training in computer and information sciences.

Simultaneously, investigators attempted to developed tools that acquired expertise directly from domain specialist. Initially, these KA tools were also general. In the mid-1980s, these general KA tools were followed by a second generation of KA tools that were specific to particular problem tasks—for instance, to classification, configuration, or scheduling. Even if the scope of the tool is restricted to one problem task, however, nonprogrammers may have difficulty using the tool [Marcus, 1988]. A third generation of even more specialized KA tools was therefore developed. Researchers started to experiment with domain-specific KA tools. Such tools are designed such that domain specialists can use well-known domain concepts in the tool dialog [Eriksson, 1992; Musen *et al.*, 1987].

Domain-oriented KA tools can provide effective support within their area, because they draw their power from built-in domain concepts that users can identify easily. However, the development of such KA tools is costly, since the amount of programming required to implement such domain-specific tools is large in comparison to the scope of the tools' services. There are three fundamental approaches to this problem: (1) balancing tool generality versus domain-orientation to achieve a reasonable trade-off between utility and cost, (2) improving further general tools, and (3) reducing the cost of developing domain-oriented KA tools (e.g., through technological means).

We have chosen the third approach. Our goal is, thus, to make it easier for developers to design and implement tools tailored for their needs. Meta-tools can help developers to create new domain-specific tools as well as to custom-tailor existing tools for a domain. There are two principal roles for meta-tools in this approach: (1) to address the software-engineering problem of developing (and specializing) target tools, and (2) to support the target-tool design and specification process. In addition to meta-tools, development *methodologies* that incorporate specialization of development tools can help the developer to control the development process and to reduce its cost.

One feature that distinguishes KA tools from other development tools is the intended tool user. KA tools are primarily intended for use by domain specialists, whereas code-generating software-engineering tools are generally designed to be use by developers with programming knowledge. So far, we have primarily worked with KA tools for knowledge-based systems. Nevertheless, several of our results can be generalized to other types of software-development tools.

Meta-Views

The most important aspect of a meta-tool is the specification model of the target tools that it provides to the developer. The meta-view adopted by the meta-tool guides the specification process, and determines the scope of the meta-tool. Meta-tools can differ substantially, depending on what aspects the meta-tool developer chooses to emphasize in the meta-view. Preferably, the meta-view should in some way reflect the way that developers think about the target tools, and should provide a natural way of specifying target tools. Several groups of meta-views can be identified.

The Method-Oriented View

The *method-oriented* view provides a framework for describing the problem-solving method to be used in the final application in a way that makes the description useful for generation of KA tools. Meta-tools implementing a method-oriented view produce target KA tools from a partial instantiation of a generic problem-solving method (e.g., planning, scheduling, or troubleshooting methods). Target KA tools are fully instantiated according to the expertise required by the problem-solving methods for performing their tasks. For example, the developer can instantiate a planning method by providing descriptions of *actions* (and their preconditions as well as ramifications), *constraints*, and *goals*. A domain-specific KA tool that allows specialists to enter and edit skeletal plans can be produced from such an instantiation of the planning method by a meta-tool supporting the planning method. Typically, meta-tools adopting a method-oriented view incorporate some form of a priori design of the target tools. One of the advantages of the method-oriented view is that the instantiation of a generic method structures the development process and guides the developer. Another advantage is that the target tool can be developed rapidly if a problem-solving method for the application is known.

There are, however, drawbacks of the method-oriented view. A significant problem is that the meta-tool is restricted to one particular problem-solving method. KA tools that acquire knowledge for other problem-solving methods, including KA tools for domains where the problem-solving method supported is unsuitable, cannot be specified using the method-oriented approach. Another problem is that the type of KA tools produced for a particular domain is fixed (i.e., it is possible to have only a one-to-one correspondence between an instance of a problem-solving method and its KA tool, due to the

a priori KA tool design). Adapting a meta-tool for another problem-solving method is currently a laborious task that may involve a major redesign of the meta-tool.

The Abstract-Architecture View

The *abstract-architecture* view is based on an architectural model of the target tool. In this approach, the developer specifies components of the target KA tool, such as the user interface, the internal representation, and the generator for target code. In other words, to create a target KA tool, the developer has to instantiate each of the components in the KA tool architecture and to link together the components. (Naturally, this task requires a prior analysis of the domain and of the requirements on the KA tool.)

Meta-tools adopting this meta-view produce KA tool implementations from abstract specifications of target KA tool components. In a way, the abstract-architecture view is similar to specification languages found in compiler compilers (e.g., Yacc and Bison). The abstract-architecture view differs from the method-oriented view in that it focuses on the target tool rather than on the application program under development. The abstract-architecture view provides more flexibility for the developer than does the method-oriented view, because many tools potentially can be specified for one domain.

The major advantage of the abstract-architecture view is that target KA tools can be specified independently of the problem-solving method adopted. Hence, the meta-tools do *not* have to rely on specific problem-solving methods (or on any other class of domains). There are, however, other limitations: The abstract-architecture view imposes restrictions on the types of target tools that can be specified. For instance, a meta-tool supporting architectural components for graphical editing and browsing cannot easily be used to produce debugging tools (which require a completely different set of architectural components). Another disadvantage of the abstract-architecture view is that the developer needs to be aware of the architecture of the target KA tools, which knowledge is not required for the method-oriented view (where the developer is required to know only the problem-solving method).

The Organizational View

The *organizational* view captures the intended organizational context for the system under development. The idea is to derive the target system's role from an organizational model (e.g., an enterprise model) and to identify the task of the system from its role. When the task of the system has been established, it can be used together with the organizational model to specify target KA tools to a meta-tool. To specify a target KA tool according to the organizational view, a developer must (1) identify the actual organizational structure from a library of typical organizations, and (2) indicate the relevant position and role in the organization for the system. In essence, the organizational perspective is an approach to create a job description for the system.

The organizational view provides a broader perspective on KA tool specification than do the method-oriented and abstract-architecture views. The broad perspective is an advantage of the organizational view, since it helps to clarify how the system is to be used and to make this information available to the meta-tool. Another advantage of the organizational view is that organizational information is often easily available and can be provided by nonprogrammers. One of the problems with the organizational view, however, is that it is not clear whether such a model is sufficient to specify a KA tool completely. Additional information, such as identification of appropriate problem-solving methods and other technical issues, might be needed to produce automatically or semiautomatically target KA tools that can be used by people in the organization (i.e., nonprogrammers) to develop the system. A pure organizational model would not provide sufficient information, but an extended organizational model might be practical for the tool generation.

The Ontological View

The *ontological* view is based on the idea that domain concepts and relationships can be used for generation of domain-specific KA tools that incorporate such concepts and relationships. Concept definitions in the ontology can be used as a basis for automated generation of domain-oriented editors in the KA tool. Target KA tools can then be used to acquire details about the domain concepts. For example, instances of domain-specific classes in the ontology can be entered and edited in the target KA tool by developers and by domain specialists. To complete a target KA tool, however, the developer might have to provide additional information in the ontology (e.g., information about how to edit certain concepts). The ontological view differs from the previously mentioned meta-views in that it focuses on declarative structures required in the application system.

Composed Meta-Views

An important question is whether we can combine several meta-view such that we avoid some of the disadvantages of particular meta-views. For instance, a combination of the method-oriented and the abstract-architecture views can potentially render a meta-view that provides the guidance of a predetermined problem-solving method and the capability to custom-tailor the target tool (e.g., for individual users). There are, however, several conceptual and technical obstacles to implementation of composed meta-views. For example, meta-views can be partially incompatible, and changes to specifications made according to one meta-view might affect—and even invalidate—other specifications according to other meta-views.

Meta-Tools

There are several meta-tools that implement the meta-views described in the previous section. We shall briefly examine four different meta-tool imple-

mentations, and shall relate them to their meta-views.

PROTÉGÉ

PROTÉGÉ [Musen, 1989a; Musen, 1989b] is a meta-tool that adopts a method-oriented view. PROTÉGÉ supports a particular problem-solving method for planning (skeletal-plan refinement), which also is the basis for the meta-view in PROTÉGÉ. Historically, PROTÉGÉ was abstracted from a domain-specific KA tool (OPAL) that acquires skeletal plans, or protocols, for cancer therapy. PROTÉGÉ incorporates an a priori design of target KA tools that is similar to the design of OPAL. The meta-view in PROTÉGÉ comprises concepts related to skeletal planning—for example, *planning entities* (which are processes that take place over finite periods of time), *task-level actions* (which are operations that control the planning entities and modify the plan during run time), and *input-data* specifications.

To build a KA tool with PROTÉGÉ, the developer must instantiate the skeletal-planning method supported by PROTÉGÉ for the domain in question. This instantiation involves describing planing entities, task-level actions, and input data in detail. PROTÉGÉ produces a target KA tool, which can be used by domain specialists to enter and edit skeletal plans, from the instantiated problem-solving method. In turn, the target KA tool produces the application system from the skeletal plans entered.

An important achievement of PROTÉGÉ is that it demonstrated how meta-tools can be used to instantiate KA tools from descriptions of problem-solving methods (i.e., PROTÉGÉ demonstrated the feasibility of the method-oriented view). Nevertheless, the principal drawback of PROTÉGÉ is inherited in its meta-view—the meta-tool is limited to one problem-solving method.

DOTS

DOTS [Eriksson, 1991] is a meta-tool that is based on the abstract-architecture view. Like PROTÉGÉ, DOTS is abstracted from a domain-oriented KA tool, but DOTS focuses on the architecture of the target tool, rather than on the problem-solving method of the application system. DOTS generates target KA tools from architectural specifications. Furthermore, DOTS assumes that the target tools conform to a particular architecture scheme (i.e., DOTS cannot be used to develop any type of software; it is tailored for development of graphical KA tools).

The meta-view in DOTS comprises (1) a variety of editors that can be custom-tailored to edit domain-specific structures, (2) a specification language for the internal representation (which represents what is entered in the editors internally) and other data structures for the target KA tool, (3) a set of update rules that can be configured to ensure consistency between the internal representations and the editors in the user interface, and (4) a set of transformation rules that is used to produce target code from the representation internal to the KA tool. To develop a KA tool with DOTS, the developer must analyze the domain and design a KA tool architecture for the domain, enter specifications for the domain-specific editors in DOTS, specify the internal representation for the target KA tool, declare the relationship between the editors and the internal representation in the form of update rules, and write transformation rules for code generation from the internal representation. DOTS produces a target KA tool from these architectural descriptions.

DOTS demonstrated how an abstract-architecture view can be implemented in a meta-tool. Unlike PROTÉGÉ, DOTS is not restricted to a particular problem-solving method or to any other domain class. DOTS, however, is restricted to a particular type of architecture for target KA tools.

SIS

Another meta-tool that implements an abstract-architecture view is SIS [Kawaguchi *et al.*, 1991]. SIS differs from DOTS in that it is designed for generating interview-based KA tools (i.e., KA tools that conduct a question-and-answer dialog with domain specialists to elicit domain information and knowledge), rather than graphical KA tools based on interactive editing for which DOTS is designed. The components of the architecture scheme supported by SIS, therefore, are different from those found in DOTS.

Spark

Researchers at Digital Equipment Corporation (DEC) have explored the organizational view as a basis for meta-tools. They have developed Spark, a meta-tool that implements the organizational view [Klinker *et al.*, 1991].

To implement a KA tool using Spark, the developer must identify the organizational type (e.g., manufacturing industry, service organization, or government), identify the role of the system in that organization using a diagram of typical organizations, and assemble a performance system using reusable program mechanisms from a library. Spark configures an appropriate KA tool from the description of program mechanisms and the information requirement for each of the relevant mechanism. The original Spark approach has been modified; the group at DEC is now considering mechanisms with a finer granularity.

Spark is part of a tool set that contains two other tools: Burn and FireFighter. Burn is the run-time system that controls the knowledge-acquisition session and invokes appropriate KA tools. FireFighter is a debugging tool that helps developers and domain specialists to debug and maintain application systems developed.

Programming Languages as Meta-Tools

General programming languages (e.g., C, Pascal, and ADA) also can be regarded as meta-views and their compilers can be seen as meta-tools, since they can be used to implement target KA tools. Programming languages, however, provide neither much support for tool implementation, nor any high-level constructs for tool specification (especially for interac-

tive tools with graphical user interfaces). The use of programming languages can certainly provide flexibility in the tool design, but the implementation cost is often too high. Nevertheless, programming languages can play a role in implementation of tool functions that cannot be specified with an available meta-view.

Summary and Conclusions

Domain-specific development tools, including domain-oriented KA tools, are often reported to be more successful than are their general counterparts. Consequently, specialized development and KA tools are emerging. Since the development of such custom-tailored tools is relatively laborious given their restricted scope, researchers have experimented with meta-tools that support the design and implementation of domain-specific tools. Although it is preferable that meta-tools be domain-independent, their generality must be restricted if they are to be practicable and supportive. One such restriction is the class of target tools the meta-tool produces.

A meta-view is the specification strategy for target tools adopted by the meta-tool. The *method-oriented* view focuses on a problem-solving method that is applicable to many domains. The developer specifies domain-oriented target tools by instantiating a problem-solving method for the domain in question. The *abstract-architecture* view, on the other hand, focuses on the architecture of the target tool. In this approach, domain-oriented tools are specified through instantiation of architectural components (e.g., graphical editors, internal structures, and sets of transformation rules). The *organizational* view provides a model of generic organizations in which the role of the application system can be identified. Such roles are used as basis for generation of target tools.

The meta-views examined in this paper represents complementary approaches to specification of target tools. Since each meta-view has advantages and disadvantages, the choice of meta-view depends largely on the requirements on the target tool, development philosophy, and personal preferences. Ideally, meta-tools should support target-tool specification according to multiple paradigms.

With appropriate meta-tools, development of *application-specific* tools (rather than domain-specific) custom-tailored to particular development situations can be made feasible. Target tools can be changed during the course of the project to support different project stages in different ways. For example, target tools can serve as specification tools and then as maintenance tools, as the project evolves.

We are currently developing a meta-tool (PROTÉGÉ II) that will support a combination of meta-views [Puerta *et al.*, 1991]. PROTÉGÉ II will support two different development tasks simultaneously. One part of the emerging PROTÉGÉ II system will allow the developer to create basic performance systems by configuring tasks and problem-solving methods from a library of reusable components, the other of part PROTÉGÉ II is concerned with generation of domain-oriented KA tools (which are used for acquiring knowledge from domain specialists for the basic performance systems). For the KA-tool generation component, we are currently considering a combination of the abstract-architecture and ontological views. Since PROTÉGÉ II is also intended for configuration of tasks and problem-solving methods, the combined meta-view will incorporate ideas from the method-oriented view also.

Acknowledgments

This work has been supported in part by grants LM05157 and LM05208 from the National Library of Medicine, by a gift from Digital Equipment Corporation, and by scholarships from the Swedish Institute, from Fulbright Commission, and from Stanford University. We are grateful to Angel Puerta for comments on drafted versions of this paper and to Lyn Dupré for editorial assistance.

References

Eriksson, Henrik 1991. *Meta-Tool Support for Knowledge Acquisition.* PhD thesis 244, Linköping University.

Eriksson, Henrik 1992. Domain-oriented knowledge acquisition tool for protein purification planning. *Journal of Chemical Information and Computer Sciences* 32(1):90–95.

Kawaguchi, Atsuo; Motoda, Hiroshi; and Mizoguchi, Riichiro 1991. Interview-based knowledge acquisition using dynamic analysis. *IEEE Expert* 6(5):47–60.

Klinker, Georg; Bhola, Carlos; Dallemagne, Geoffroy; Marques, David; and McDermott, John 1991. Usable and reusable programming constructs. *Knowledge Acquisition* 3(2):117–135.

Marcus, Sandra, editor 1988. *Automating Knowledge Acquisition for Expert Systems.* Kluwer Academic Publishers, Norwell, Massachusetts.

Musen, Mark A.; Fagan, Lawrence M.; Combs, David M.; and Shortliffe, Edward H. 1987. Use of a domain model to drive an interactive knowledge-editing tool. *International Journal of Man-Machine Studies* 26(1):105–121.

Musen, Mark A. 1989a. *Automated Generation of Model-Based Knowledge-Acquisition Tools.* Morgan-Kaufmann, San Mateo, California.

Musen, Mark A. 1989b. An editor for the conceptual models of interactive knowledge-acquisition tools. *International Journal of Man-Machine Studies* 31(6):673–698.

Puerta, Angel R.; Egar, John W.; and Musen, Mark A. 1991. Automated generation of adaptable knowledge-acquisition tools with Mecano. Technical Report KSL-91-62, Knowledge Systems Laboratory, Stanford University, Stanford, CA.

SOFTWARE DESIGN AS A PROBLEM IN LEARNING THEORY

(A Research Overview)

Leona F. Fass

Introduction: Background and Motivation

Our interest in automating software design has come out of our research in automated reasoning, inductive inference, learnability and algebraic machine theory. We have investigated these areas extensively, in connection with specific problems of language representation, acquisition, processing and design.

In the case of formal context-free (CF) languages we established existence of finite learnable models ("behavioral realizations") and procedures for constructing them effectively. We also determined techniques for automatic construction of the models, inductively inferring them from finite examples of how they should "behave". These results were obtainable due to appropriate representation of domain knowledge, and constraints on the domain that the representation defined.

It was when we sought to generalize our results, and adapt or apply them, that we began investigating the possibility of determining similar procedures for constructing correct software. Discussions with John Cherniavsky, Dick Hamlet and Elaine Weyuker led us to examine testing and verification processes, as they are related to inference, and due to their considerable importance in correct software design. Motivating papers by Cherniavsky [1], Hamlet [3], Weyuker [4] and also, Fetzer [2], led us to examine these processes in some depth.

Here we present our approach to those software design issues raised in [1-4], within our own theoretical context. We describe our results, relative to those of [1-4] and conclude that they do not compare unfavorably.

Our Approach To Software Design

We approach problems of software design as examples or applications of a general learning theory. Our perspective is logical and algebraic: to us, a program or system fulfilling a specification S is "just like" any other realization of a specified behavior. The process of constructing software to perform a particular function or set of tasks, thus is an instance of synthesizing a behavioral realization. The testing of given software for incorrectness, or its verification as correct, are cases of checking a potential model, or realization, against its behavioral domain. If it is determined to exhibit all "good behavior" (positive domain data, as specified by S) and no "bad behavior" (negative data, i.e., the *complementary* domain elements, relative to S) the software is then established as correct.

Within our theoretical framework, successful software design requires analysis of desired behavior for identification of its essential components, and a means of defining--often through constraints--the domain in which the behavior lies. This knowledge must be represented and conveyed to the design system: an algorithm or technique for converting the knowledge into an implementation. Should designed software be *given*, then the knowledge might be conveyed to a testing/verification system to determine correctness of the design. If incorrectness were detected, errors could be removed and flaws repaired. The theoretical system need only reiterate these steps until it conclusively determined the software to be defect-free.

In each of these aspects of software design, our theory assesses as successful a process that is proven to terminate effectively (many would also demand efficiency), determining correct software as its end-product. This implies that all possible behavior must be conveyed finitely; that algorithms and techniques for construction, testing or verification of software operate in finite time and space; and that each process concludes, producing a resultant finite behavioral model.

If the above can be achieved it is a small step from effective determination of correct software to its automated determination or, design. We need only implement the algorithm or technique for the software construction, testing or verification, to create an automated "design system". Then we need only define an appropriately characterizing finite selection of behavioral data that the "system" may use to automatically determine a correct software design. To do so, we might adapt those techniques we devised to find correct language models [5-8], so that instead they produce software that behaves correctly, as specified.

Once a "design system" is implemented, it should be possible for an application specialist to provide it with domain-specific behavior examples. The system should then observe and generalize, to automatically determine software that realizes, or produces, the correct domain behavior in its entirety. At first, this appears to work very well, in theory.

However, our theoretical perspective leads us to examine software design problems somewhat more carefully, relative to those algebraic, constrained problem domains within which we obtained our initial learning theory results. We next describe some of the relationships between our theory and actual practice.

Results, "Results" and Conclusions

While there are, indeed, many similarities between theoretical learning problems and those encountered in practice, what we mainly find is that the constraints that make problems solvable in *theory* do not, in *practice*, generally apply.

We began this research overview by describing our theoretician's perspective, and our interest in adapting or applying our specific learning theory results to the case of

(automated) software design. Within the framework of theory, we noted that software design is "just like" any other modelling process. E.g., if we can infer a grammar generating a language from suitable linguistic examples then, surely, we can infer a program to produce that same language, and be certain that it is correct.

All of the general results in learning theory that come out of our specific CF language learning research were made possible by appropriate knowledge representation, and domain constraints. These enabled us to determine finite realizability of the CF languages and, also, the conclusive effective testability of potential language models. When sufficiency of testing is established, and tests conclusively detect no incorrectness, we establish correctness of a model. We call this "verification by default" [6-9].

In the case of language learning, we were able to establish an inference/testing/verification paradigm [6-10] that could result in automatic design of language models, obtainable in a number of ways. We showed that if the language has a model inferable from a finite sample of positive domain data ("good behavior") then a potential model could be conclusively, effectively tested and thus might be verified, by default, as correct. What we established was that the domain sample of positive data sufficient for inference defined a similar sample of positive and negative data ("good and bad behavior") that was sufficient for conclusive, effective tests.

As Hamlet noted in [3] and in our discussions, and as we have confirmed, these results are dependent on characterizing all necessary behavioral information in a finite way. (Our domain constraints gave us finite realizability and decidable membership queries: we could determine what was good behavior *vs* what was not [6-10]).

While in any typical software design environment our domain constraints and conditions do not apply, we believe our theoretical results compare, not unfavorably, with those of other theoreticians. Cherniavsky [1] noted testing can do more than detect errors in software, and we showed one can test to show software is correct. Fetzer [2] claimed verification was "impossible" and we showed inferable models could be testable, and verified automatically, by default. Weyuker [4] described inference-based testing to establish an approximate method of determining equivalence of a program and its specification. We concur and believe our logical and algebraic approach, and some domain-specific imposed constraints, will result in approximately automated software design. This will improve upon techniques currently in practice.

REFERENCES

[1] Cherniavsky, J. C., "Computer Systems as Scientific Theories: A Popperian Approach To Testing", *Proc. of the Fifth Pacific Northwest Software Quality Conf.*, Portland (Oct. 1987), pp. 297-308.

[2] Fetzer, J. H., "Program Verification: The Very Idea", *CACM*, Vol. 31 (1988), pp. 1048-1063.

[3] Hamlet, R., "Special Section on Software Testing", *CACM*, Vol. 31 (1988), pp. 662-667.

[4] Weyuker, E. J., "Assessing Test Data Adequacy through Program Inference", *ACM Transactions on Programming Languages and Systems*, Vol. 5 (1983), pp. 641-655.

Relevant Publications and Presentations by the Author

[5] Fass, L. F., "Remarks on Inductive Inference and Testing", presented at the *Association for Symbolic Logic 89-89 Annual Meeting*, University of California, Los Angeles, January 1989. Abstracted in the *J. Symbolic Logic*, Vol. 55, No. 1 (March, 1990), p. 374.

[6] Fass, L. F., "A Common Basis for Inductive Inference and Testing", *Proc. of the Seventh Pacific Northwest Software Quality Conf.*, Portland, (Sept. 1989), pp. 183-200.

[7] Fass, L. F., "Acquiring Knowledge by Positive or Negative Means", presented at the *Association for Symbolic Logic 90-91 Annual Meeting*, Carnegie Mellon University, January 1991. Abstracted in the *J. Symbolic Logic*, Vol. 57, No. 1 (March 1992) pp. 356-357.

[8] Fass, L. F., "Learning Through Inductive Inference or Testing", *Proc. Florida Artificial Intelligence Research Symposium, Conf. on Machine Learning*, Cocoa Beach, (April 1991), pp. 176-180.

[9] Fass, L. F., "Inference, Testing and Verification", presented at *Ninth International Congress on Logic, Methodology and Philosophy of Science* and *Logic Colloquium 91*, Section on Foundations of Logic, Mathematics and Computer Science, Uppsala, Sweden, August 1991. Abstracted in *Congress Volume I*, p. 193.

[10] Fass, L. F., "Perfect Learning (More or Less)", to be presented at the 1992 Meeting of *The Society For exact Philosophy*, University of Southwestern Louisiana, Lafayette, May 1992. Extended version in preparation.

Leona F. Fass received a B.S. in Mathematics and Science Education from Cornell University and an M.S.E. and Ph.D. in Computer and Information Science from the University of Pennsylvania. Prior to obtaining her Ph.D. she held research, administrative and/or teaching positions at Penn and Temple University. Since then she has been on the faculties of the University of California, Georgetown University and the Naval Postgraduate School. Her research primarily has focused on language structure and processing; knowledge acquisition; and the general interactions of logic, language and computation. She has had particular interest in inductive inference processes, and applications/adaptations of inference results to the practical domain. She may be reached at

Mailing address: P.O. Box 2914
Carmel CA 93921

Towards Automation of User Interface Design

Rainer Gastner Gerhard K. Kraetzschmar Ernst Lutz

Research Group Knowledge Acquisition
Bavarian Research Center for Knowledge Based Systems (FORWISS)
Am Weichselgarten 7, 8500 Erlangen, Germany
e-mail: gastner@forwiss.uni-erlangen.de

Abstract

This paper suggests an approach to automatic software design in the domain of graphical user interfaces. There are still some drawbacks in existing UIMSs which basicly offer only quantitative layout specifications via direct manipulation. Our approach suggests a convenient way to get a default graphical user interface which may be customized and redesigned easily in further prototyping cycles.

1 Introduction

The automation of software design becomes more powerful if the target systems generated are limited to a certain domain. The domain addressed in this paper is the class of graphical, highly interactive systems for accessing data of specified data structures by end users. The focus of this paper is further restricted. It concentrates on the automation of the design of a graphical user interface (GUI) for these systems.

Building GUIs with GUI toolkits or user interface management systems (UIMSs) is still a laborious, time-consuming task even if it is supported by direct manipulation facilities [6]. The basic problems we identified are the following:

- The GUI designer has to decide which graphical element is appropriate for a desired interaction, i.e. given a data structure and data type descriptions of the elements to be accessed and a set of GUI elements the designer has to perform a mapping between the data structure and the GUI elements.
- With direct manipulation an initial GUI may be built but if the data structure or the data types are changed the manual adaption of the GUI is arduous. According to the changes of a data structure the extent of the redesign task may cause pretty much effort.
- Due to the lack of adopted GUI design guidelines, for similar data structures in different applications a different GUI may exist which is contradictious to user interface consistency [10].

The approach introduced in this paper to address these problems is the automatic generation of GUIs from a high level specification. This generation is performed by a knowledge-based meta-tool which is used by a GUI designer. Questions which have to be tackled include the following: (1) To what extent can the designer be supported in the specification task? (2) What kind of user interface should the meta-tool have. (3) Which kind of knowledge is domain invariant and which is application specific (and therefore needs to be entered by the designer)? (4) Which set of default design decisions are adequate?

Our approach to answer these questions is based on the following idea: The designer specifies data structures, data types and operations which the user of the target system has to perform with an user-friendly GUI. Corresponding GUI elements realizing these operations are associated automatically and the GUI is generated. The designer in turn refines the GUI by interactivly customizing the meta-tools association and specifying *qualitative* layout constraints. This approach facilitates users who have no knowledge about interface programming to construct a GUI easily. Since the GUI of a meta-tool itself is in the domain our approch is applicable for the design of meta-tool's GUI as well.

In section 2 the addressed domain is introduced in more detail. Section 3 discusses the problems of configuration and generation of the target systems. In section 4 our approach is described to solve these problems. Section 5 compares our approach to related work and section 6 gives some concluding re-

marks and perspectives on future work.

2 Domain

The domain our meta-tool addresses is the class of GUIs that allow the access of specified data structures whose elements are characterized be specific data types. The access comprises additon, deletion, modification, selection and browsing of data structures and instances.

There exist rather different interpretations of what the notion GUI should mean [6]. In our meta-tool the GUI is built with a set of objects which have a description of a graphical presentation and methods to handle the display presentation and the communication with the underlying window system. Examples are buttons, settings or text fields. No other functionality is added to the GUI. The GUI objects are described within an object-oriented class hierarchy adopting inheritance. This is the common approach how state-of-the-art GUI toolkits and UIMSs are realized [6].

Our meta-tool produces specializations of classes in a class hierarchy provided by the GUI toolkit *LispView* [1] and instantiation methods. LispView provides an interfaces between Sun CommonLisp and OpenWindows. The same structure is generated by the GUI devolopment system *OpenWindows Developer's Guide* [3].

3 Problem Description

The design of a meta-tool for automating the design of GUIs from specifications of data structures, data types and operations raises some questions which mainly influence the meta-tool design decisions:

- Which kind of knowledge has to be represented to support the generation and which kind of knowledge representation should be used?
- Which part of the knowledge is domain specific but application invariant and which part is application specific?
- What kind of default configuration decisions makes sense? Can specific subdomains be identified for which specific configuration macros may be used?
- What is the most efficient way to enter geometric layout specifications?
- Since an initially generated GUI in most cases does not meet the end user's whishes rapid prototyping facilities for iterative refinement and customization is needed.
- The specification facility must allow only consistent specifications, i.e. the designer's specification has to be syntactically and semantically correct and the generator will produce a GUI inside the domain. How can we support specification consistency?

The following section discusses our approach towards an automation of the GUI design addressing the questions given above.

4 Approach

State-of-the-art UIMSs mainly deal with a user-friendly composition of the GUI. From this point of view only the syntactical aspects in building GUIs are addressed. But naturally GUIs are built for user interactions which have certain semantics. For instance, when the GUI designer using a direct manipulation UIMS selects a button and arranges it in the target interface via mouse dragging he knows the reason why he selects a button and which operation should be performed by clicking on the button. The GUI components are nothing else than graphical presentations of abstract interactions. The mapping from the semantics of these interactions to corresponding GUI elements is the main task of an GUI designer.

Our approach for specifying GUIs starts from a semantic point of view and focuses on this mapping. The GUI designer does not specify a composition of the GUI components itself rather than the interactions the GUI components shall be used for. That means the focus of the specification is not *how* to present interactions on the screen but *what* kind of interactions shall be established. The interactions we consider are the access operations specified in section 2. The mapping from the interaction specification to the GUI components is done by the meta-tool automatically. In a further step the designer may customize the generated GUI either by changing the mapping or specifying additional qualitative layout constraints.

4.1 Configuration Process

In this section the configuration process is discussed. Figure 1 provides an overview of the configuration steps. The actions the designer has to perform are

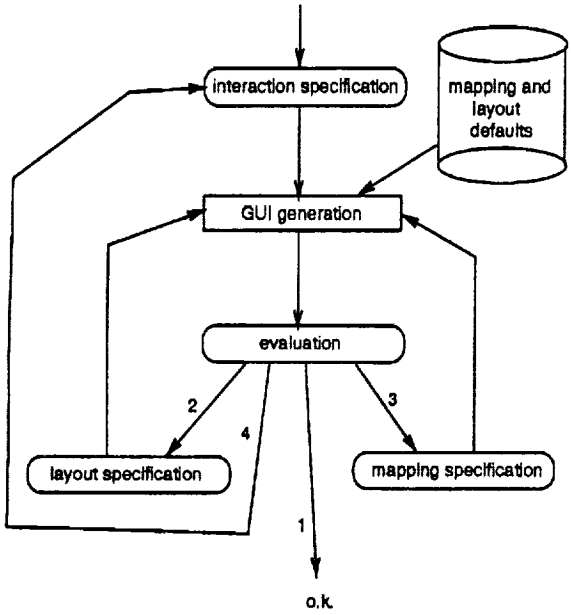


Figure1: iterative GUI configuration process

specification and evaluation represented as round-cornered boxes. The meta-tool activity (the generation of the GUI) is represented as a rectangular box.

The designer starts with the specification of the desired interactions on data structures. Then an initial GUI is generated by the meta-tool using default mapping and layout configurations stored in a knowledge base (see section 4.2). The initial generation has to be evaluated by the designer. Then one of the following four choices may be made:

1. The designer agrees with the generated GUI and the configuration process is finished.
2. The designer specifies qualitative geometric layout constraints to rearrange the GUI components on the screen.
3. The designer alters the mapping between the specified interactions and the corresponding GUI component.
4. The designer manipulates the interaction specification, e.g. a new element is added to a data structure.

In case of a new or re-specification a new generation cycle starts. The order given above implies the extent of the GUI redesign in a cycle after evaluation. Choice 2 affects only the geometric position of GUI elements, choice 3 affects the presentation of an interaction, and choice 4 affects the interaction itself. Explorative rapid prototyping by iterating the configuration cycles is supported conveniently, since the designer starts with a specification of abstract interactions omitting GUI aspects in the initial phase. In following cycles he can customize presentation aspects very quickly or redesign the interactions.

Since end users are supposed to design the GUIs the meta-tool must provide user-friendly graphical interfaces itself. To support specification consistency, the specification is menu-driven as far as possible. Menus with appropriate selections may be offered which is further discussed in section 4.2. An interesting issue is that the GUI of the meta-tool to enter the specification is itself in the domain of the meta-tool. Since the meta-tool allows to use the specification languages directly without the corresponding GUI[1], the GUI for the meta-tool can be generated by the meta-tool itself.

4.2 Configuration Knowledge and Representation

This section deals with the knowledge needed to automate the GUI configuration. We distinguish two classes of knowledge. Knowledge is needed to support an efficient user-friendly specification and to generate a GUI with an minimal specification. This kind of knowledge is application-invariant and refered as *domain-specific* (in the GUI domain). On the other hand *application-specific* knowledge must be enterd by the designer to build an GUI for a set of certain interactions. The following two subsection discuss these two knowledge classes.

4.2.1 Domain-specific knowledge

The following listed knowledge categories are stored in the meta-tool's knowledge base in order to support specification and generation. Note that this is mainly knowledge *about* the possible application-specific knowledge (e.g. possiple types of layout constraints) and therefore meta-knowledge.

- **model of target architecture**; the structure of the code generated by our meta-tool is given by the code structure the Developer's Guide for LispView interfaces [3] generates.
- **a library of interaction types and data types**; interaction types include read and write access to data and selection of data. Currently the library of data types includes enumeration, character, real, integer, string, symbol, and object-class.

[1] Otherwise there would be meta-tool tower never ending.

- **a library of GUI elements**; this library is given by the used GUI toolkit LispView [1].
- **mapping of interaction specifications to GUI elements**; the mapping is stored as a matrix in which for certain conditions made in a data type specification a set of possible GUI components is associated. The GUI component selected by default is marked (see also section 4.4.
- **library of layout constraints**; currently we have realized 36 layout constraint types which are hierarchically organized and offered in menus. Furthermore, there exits a layout constraint construction facility for the meta-tool designer to implement additional constraint types based on a combination of types from a basic set.
- **standard configurations**; see section 4.4.

The domain-specific knowledge is stored in ASCII-files in special representation languages. The files may either be edited directly by a text editor or be generated from graphical specifications. An interpreter reads these files and maps the external representation to internal objects.

4.2.2 Application-specific knowledge

As shown in figure 1 there are three specification possibilities providing input for the generator.

- **interactions**; the specification comprises the type of operation and the data type to be accessed. The data type is specified separately. Thus more than one interaction may access data of the same data type in different ways. The example below shows the declarative specification generated from the graphical specification environment. A manipulation interaction is specified on data of an integer slot. The value range is restricted between 100 and 500, the slot is single-valued, and the value must be unique and entered.

```
(def-interaction
        :id 'engine-number-manipulation
        :operation 'manipulation
        :data-type 'engine-number-type)
(def-integer
        :id 'engine-number-type
        :equalorgreater 100
        :lessorequal 500
        :mincard 1
        :maxcard 1
        :unique T)
```

The declarative specification languages may also be used directly by the designer. Both interactions and data types are offered in menus to the designer. The menus are configured dynamically according to certain specification constraints; e.g. the following constraint may not be violated in the example above: `(lessorequal mincard maxcard)`.

- **association of interactions and GUI components**; if the designer does not agree with the meta-tool's association he may select another association or more than one associations for a given interaction from a menu. The menu items consist of all GUI components which are acceptable presentations for the interaction asserting a consistent specification. If the designer associates more than one GUI component to an interaction, the interaction is presented in different fashions in the GUI. For instance, the interaction in the example above may be presented as numeric field or a slider. The meta-tool would select the numeric field by default.
- **layout constraints**; the qualitative layout constraints may be specified using a declarative specification language or a GUI generating sentences of this language. The following example demonstrates the power and user-friendlyness of our layout mechanism:

Let $B_1, B_2, \ldots, B_5$ be boxes which shall be arranged as follows: B_4 and B_5 shall be at the bottom of the layout frame; B_1 shall be in the upper left corner of the layout frame; and B_3 shall be over B_2 and B_5. This is expressed as follows:

(bottom-margin B_4 B_5)
(upper-left-corner B_1)
(over B_3 (B_2 B_5))

Entering the first layout constraint via the specification GUI B_4 and then B_5 would by selected with the mouse on the screen and then the constraint `bottom-margin` would be selected from the menu.

4.3 Layout computation

Each GUI element has a rectangular bounding-box which provides the size for the layout generator. The 36 layout constraints are one-dimensional geometric relationships between these boxes. N-ary relationships are resolved into binary ones which are connected with a conjunction. The corners of the boxes are represented by variables and the constraints are always inequations of the following form:

$$\alpha_i \leq x_j - x_i \leq \beta_i$$

These unequations can be solved using a longest-path

algorithm suggested in [11]. If there are inconsistencies in the specified constraint set our algorithm retracts contradictious constraints. The boxes are arranged fulfilling the specified constraints and are positioned in the upper left corner of the layout frame. In a second cycle overlapping boxes (this may occur if the constraint set is not restrictive enough) are solve by adding additional contraints with disjunctions: A box B_1 and a box B_2 do not overlap if (beside $B_1 B_2$) $\vee$ (beside $B_2 B_1$) $\vee$ (over $B_1 B_2$) $\vee$ (over $B_2 B_1$) holds. Since there may be a huge number of layout configurations solving the constraint set without overlapping the layout algorithm gets a certain time for processing (e.g. three seconds). The algorithm generates a set of solutions and then selects the best solution when the time is over. The selection criteria adopted currently is either to minimize the area of the layout frame if the size is not prespecified or to arrange the boxes with equal distances between them in a fixed layout frame.

4.4 Standard Configurations

In order to give support in the specification of GUI component associations to interactions and to select default associations we (partly) represent knowledge found in the *OPEN LOOK application style guidelines* [2]. This knowledge is stored in a matrix in this way that for each GUI component it is marked under which conditions it is appropriate and if it should be selected by default. Furthermore, OPEN LOOK provides a unique look-and-feel for all the target GUIs and the GUI sepecification environment of our meta-tool.

It is possible to preconfigure special editor types which include a number of fixed interactions. For instance, a login editor consists always of two interactions, one for entering the user's name and one for entering the password. These two interactions are preconfigured as a symbol and string manipulation interaction. Furthermore, a layout frame with a fixed size is configured, layout constraints are specified that both GUI components (the meta-tool will associate two text fields) should be centered and the text field for the user's name should be located over the field for the password entry. The configuration is stored as subclass of a preconfigured editor class. Other specialized editors may be partly preconfigured and layouted like object editors or browsers. Preconfigured GUI classes can be dynamically added by the designer.

Adopting this configurartion library and the represented OPEN LOOK style guidelines we facilitate the generation of GUIs which have a common structure and supports GUI consistency [10].

4.5 Implementation

Our meta-tool is implemented in Sun CommonLisp, CLOS and LispView [1]. Object-oriented programming is adopted basically. The target code is generated using templates which are expanded according to the designer's specification or standard configurations. By replacing the templates it is possible to generate other GUI target code as well.

5 Related Work

In the last decade human-computer interaction and the user interfaces have become an important research field. UIMSs try to improve GUI development and support mechanisms for GUI and dialogue specification, representation and management [6] [9]. In [7] several generations of UIMSs are identified. It is predicted that future UIMSs will be knowledge-based and generate a user interface automatically using the specification of the underlying application. Our approach is a step in this direction. Currently the interactions have still to be coded by a GUI designer, but there should be a way to generate the interaction specification from application programs automatically as well.

A number of development methodologies have been suggested for user interfaces. Most of them claim explorative prototyping as our approach (see figure 1), e.g. the star life cycle suggested in [7].

User interfaces may be specified language-based with special user interface description languages, graphical-based with direct manipulation facilities or with automatic generation from interaction descriptions [9]. Since our meta-tool generates code which can be manipulated by the Developer's Guide [3] our approach combines these three possibilities which may be alternatively used.

Similar approches for automatic generation of GUIs are used in the GADGETS system [8] and the PRED system [13], but they lack qualitative layout specifications. Automatic presentation systems for information like SAGE [12] also use meta-information to select an adequate presentation style. A similar approach of default configurations of editors is applied in the meta-tool DOTS [4].

6 Concluding Remarks and Future Work

We suggested an approach towards automation of user interface design which starts from a semantic point of view. The initial specification only deals with *what* the GUI is to be built for and not *how*. Further prototyping cycles allow to customize the generated GUI *qualitatively*. Since the generated GUI code is interpretable by the direct manipulation tool Developer's Guide [3], also quantitative layouting is available and may be adopted alternativly. Since the meta-tool's GUI is in the meta-tool's domain itself a reflexive application of the meta-tool is possible.

In the project KME (Knowledge Maintenance Environment)[2] we designed a meta-tool called KME workbench [5] for generating maintenance components for knowledge bases of expert systems. A maintenance component for updating objects of an object oriented representation needs a GUI of the domain described in this paper. Thus the GUI design meta-tool is part of the KME workbench. We experienced in this project that qualitative layout specifications are very convenient and allow rapid explorative prototyping. The GUI specification environment also allows end users (e.g. knowledge engineers with only few programming experience) to build adequate GUIs easily.

We acquired GUI design knowledge from the *OPEN LOOK GUI application style guidelines* [2] which is represented in a matrix representation and allows the meta-tool to provide default configurations. Furthermore, the explicit representation can easily be changed and augmented.

Currently we work on the extension of default configurations and GUI facilities. Special editor types are identified in more specific application domains and represented. We will evaluate how the GUI specification can be acquired automatically from the underlying application. In the knowledge maintenance context we will try to generated a default dialogue control supported by a transaction management.

[2] KME was started as joint project between FORWISS and the company BMW, Munich.

References

[1] *Lisp View Programming Manual.* Sun Microsystems, Inc., 1989.

[2] *OPEN LOOK Graphical User Interface Application Style Guidelines.* Sun Microsystems, Inc., Addison-Wesley, Reading, Massachusetts, 1990.

[3] *OpenWindows Developer's Guide 1.1, User's Manual.* Sun Microsystems, Inc., 1990.

[4] Henrik Eriksson. *Meta-Tool Support for Knowledge Acquisition.* PhD thesis, Linkoeping University, Sweden, 1991.

[5] Rainer Gastner, Gerhard K. Kraetzschmar, and Ernst Lutz. Kme-workbench: a meta-tool for designing maintenance components for knowledge based systems. paper submitted to ECAI92, January 1992.

[6] H. Rex Hartson and Deborah Hix. Human-computer interface development: concepts and systems for its management. *ACM Computing Surveys*, 21(1):5–92, March 1989.

[7] H.R. Hartson and D. Hix. Toward empirically derived methodologies and tools for human-computer interface development. *Int. Journal of Man-Machine Studies*, 31(4):477–494, October 1989.

[8] Johannes L. Marais. The gadgets user interface management system. *Structured Programming*, 12(2):75–89, 1991.

[9] Brad A. Myers. User-interface-tools: introduction and survey. *IEEE Software*, 15–23, January 1989.

[10] Jakob Nielsen, editor. *Coordinating User Interfaces for Consistency.* Academic Press, London, 1989.

[11] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen.* BI Wissenschaftsverlag, Mannheim, 1990.

[12] Steven F. Roth and Joe Mattis. Automating the presentation of information. In *Seventh IEEE Conference on Artificial Intelligence Applications*, pages 90–97, IEEE, IEEE Computer Society Press, Washington, February, 24–28 1991.

[13] S. Xie and P. H. Winne. Kamit: a knowledge acquisition and maintenance interface tool. In M. H. Hamza, editor, *Expert Systems Theory and Applications*, pages 115–118, IASTED - Acta Press, Anaheim, 1988.

TOWARD DOMAIN-SPECIFIC DESIGN ENVIRONMENTS

Some Representation Ideas from the Telecommunications Domain

Sol Greenspan and Mark Feblowitz

GTE Laboratories Incorporated
Computer and Intelligent Systems Lab
40 Sylvan Road
Waltham, Massachusetts 02254
617-466-2962
greenspan@gte.com

Introduction

ACME[1] is an experimental environment for investigating new approaches to modeling and analysis of system requirements and designs. ACME is built on and extends object-oriented conceptual modeling techniques and knowledge representation and reasoning (KRR) tools [Greenspan, et. al. 1991]. The most immediate intended use for ACME is to help represent, understand, and communicate system designs during the early stages of system planning and requirements engineering.

While our research is ostensibly aimed at software systems in general, we are particularly motivated to make an impact in the telecommunications domain, especially in the area referred to as Intelligent Networks [IEEE Comm., Dec. 1988], [IEEE Comm., Feb. 1992]. Intelligent Network (IN) systems contain the software to provide services to users of a telecommunications network (e.g., call processing services, information services, etc.) as well as the software that provides the internal infrastructure for providing the services (e.g., resource management, billing, etc.). The software includes not only systems developed by the network proprietors but also by a growing group of independent service software providers.

The kind of software design problem we are interested in is at a high level. It involves, among other things, deciding where, in a distributed heterogeneous system, to locate program logic, data, and other resources; conceptually speaking, how to assign responsibilities and capabilities for carrying out the services [Greenspan 1991]. The situation is often an evolving one: given an existing situation, new requirements arise, such as the need for a new service or a new capability, and the design problem is how to (re)design the system to respond to the change.

We are quite sure that IN systems analysts and designers use a great deal of domain knowledge to make decisions about how to design an IN system to meet new requirements, and that their familiarity with the domain is a dominant factor affecting the ultimate success of the system design. The question is what that knowledge is and how it can be represented in a domain-specific environment. In this paper, we will briefly survey a few of these representation ideas and how they contribute to the goal of domain-specific software design. To the extent that these ideas are cogent applications of general software engineering principles, their essence should apply to other domains as well.

Design from domain-specific building blocks

In the telecommunications domain, there are several mandates for having a stable set of building blocks from which service software can be composed and rapidly implemented. One impetus for building blocks is the need for the industry to agree on a basic set of services and capabilities that can be assumed as universal so that services can interwork over company and national boundaries. Another impetus is that the US federal government seeks to promote fair competition by making sure that a common set of building blocks is available to all potential service developers/providers (not only to the telecommunications network proprietors).

Although the forces that motivate the use of building blocks may be largely nontechnical or quasi-technical, the emphasis on a building block approach turns out to be a valuable idea from a design point of view. It narrows the search space for solution software because all solutions must be composed from officially sanctioned building blocks. Moreover, the resultant software can be more correct, reliable, and so on, since building blocks are subject to intense scrutiny and analysis. The building block approach may appear to bring with it a loss of design

---

1 ACME is an acronym for A Conceptual Modeling Environment.

freedoms, since software is not allowed to decompose into arbitrary software components, but the premise here is that the gain in manageability of the design process is worthwhile compensation for this

The essential ideas of a building block approach are as follows. First, building blocks need to be (a) adequate to compose the desired set of services, and (b) implemented effectively in components of the systems that provide the services. Secondly, building blocks need to be reliably, efficiently, safely (etc.) implemented in the embedded system base. Thirdly, the introduction of new building blocks into the system fabric needs to be a controlled process. Suppose an organization desires to offer a new class of services that requires building blocks not already available in the system; the newly required building blocks need to be carefully identified, implemented, and tested, and importantly and nontrivially, their interactions with the old building blocks need to be taken into account.

It is important not to confuse the building block approach with the general notion of reusable components. The main idea of reuse, in its most general sense, is an asset management idea, namely that prior investment in software artifacts (code, specifications, or whatever) can be capitalized on by reusing the artifacts. If an enterprise restricts its software development to a specific domain, then the existence of domain-specific reusable components may enable one to achieve a higher degree of reuse. It has been pointed out that domain analysis is a way to achieve this (e.g., see [Arango & Prieto-Diaz, 1991]). However, this is still not the idea of building blocks. Reusable components refer to a library of assets that happen to be available to designers, while building blocks refer to the set of software components that have been designed into the system infrastructure of the operational system.

We suspect that a building block approach is already being used in other software domains and is worth making an explicit principle for building domain-specific environments. Further insights can be gained by drawing parallels between software development and other forms of manufacturing, where a set of building blocks (or "parts") are used to assemble products. Software is different in the respect that an infinite variety of "parts" can be created, which is both an opportunity and a management problem.

Domain-specific layering based on design decisions

In the telecommunications domain, standards groups are discussing a four-level IN conceptual model [Duran & Visser 1992] that organizes Intelligent Network systems in a useful way that might apply to other domains. While the model itself is not complete in any sense and is continually evolving, there are some ideas worth noting. We will not give a literal description of the four-plane IN conceptual model but rather give a rough summary and extract some of the key ideas, using vocabulary convenient for the purposes of this paper.

The layers/planes are roughly the following, from top to bottom:

1) Services -- The software applications for the end user.
2) Service Building Blocks -- As discussed above, software components that are used to compose services and which are provided by the underlying service-providing system/network.
3) Logical System Entities -- A set of standard system components (called "functional entities" by the standards groups), each of which offers methods that implement the building blocks.
4) Physical System Entities -- A set of available system components that can be developed or procured and installed in the embedded base. They are, conceptually, packages of logical system entities. Vendors build these.

These planes are usefully chosen so that the relationships between adjacent levels involve key types of design decisions. We already discussed the relationship between Services (level 1) and Service Building Blocks (level 2).

The main rationale for level 3, Logical System Entities, is that the industry needs to have a way of identifying, specifying, and integrating systems in a vendor-independent manner. Besides this motivation, level 3 also seems to be the focal point for several design concerns. Level 3 identifies the perceived infrastructure of logical, service-providing systems. Elaboration of this plane would describe the perceived standard subsystems that comprise the domain, such as making phone calls, billing, reporting error messages, and so on. This level will be quite rich with domain-specific content, representing, in effect, a model of the service-providing enterprise (discussed further below).

The relationship between Service Building Blocks and Logical System Entities (i.e., between levels 2 and 3) concern how capabilities are distributed among logical system components. The relationship between Services (level 1) and Logical System Entities (level 3) concern design decisions about what system entities are responsible for playing various roles in services.

The design decisions relating Logical and Physical System Entities (levels 3 and 4) mostly concern designing a physical system to meet nonfunctional requirements. Logical systems will have associated nonfunctional requirements (e.g., concerning performance, reliability, security, etc.) that must be met by the physical system entities. The original sources of nonfunctional requirements might actually be traceable to any of the levels. In any particular domain, one must identify and specify the nonfunctional properties that are most critical to success in that domain. Arguably, the design knowledge about how designers design physical systems to successfully meet the nonfunctional properties seems one of the most difficult subjects to formalize and automate.

This discussion of the four-plane model is intended to point out some of the more generic (high-level) design issues that might transfer across domains. The industry has

developed other, more detailed, layered models (such as the seven-layer OSI architecture), which is more intensely domain-specific to communications and less likely to transfer.

Enterprise domain knowledge

The domain-specific design environment for systems in our domain should be able to take advantage of the fact that all of these systems ultimately are part of an enterprise that provides services (either to end-users or to internal agents responsible for tasks necessary for providing the service). Given what is known about the nature of these systems, there are a lot of assumptions and constraints that can be built into the design environment.

For example, in the domain of telecommunications services, there are customers who subscribe to services. Services are tasks performed by service provider agents for customers, usually involving sensing and changing the state of objects in the customer environment and performing communication acts across a network of objects. It is further known that when a customer signs-up for (or subscribes to) a service, some service-related objects may need to be installed (e.g., a telephone at the customer premises, a wire to the customer's residence, customer information in a system database -- this is called "provisioning"). Another part of the domain is that services are performed in exchange for payment, which requires data on the use of services by customers. These and other aspects of the enterprise can be and should be part of a domain specific environment for designing systems in that domain.

One advantage to be gained by ACME from the presence of enterprise domain knowledge is that model acquisition can be supported by intelligent assistance, as in [Reubenstein 1990]. For example, since the assistant knows that provisioning is done when a new customer signs up for a service, the system can know something about what information needs to be specified (and can partially fill it in).

Designing systems in terms of the enterprise domain knowledge is much easier than working at a general systems level. General-purpose CASE environments, which offer generic concepts such as objects, properties, entities, processes, and so on, leave too large a semantic gap between the subject matter and the representation scheme. (On the other hand, we are in favor of building on general-purpose modeling concepts; see [Greenspan, et. al. 1991].) In [Greenspan 1991], we actually propose the use of an intermediate level of domain-specificity, called Service-Oriented Systems (SOSs), that takes advantage of some of the knowledge of service-providing systems in our domain but still remains relatively generic.

Process domain knowledge

The above argument for using domain knowledge can be extended to process knowledge, namely the process of designing and developing the system. This is sometimes called process knowledge or methodology modeling. Process domain knowledge deals with how the system/software artifacts are created and how they evolve. Given that we know that the artifacts we are designing belong to a specific domain (e.g., systems that provide IN services), we can specialize our view of the process that creates these artifacts. We are not creating just programs, or subsystems -- we are creating services, service-providing systems, and so on. Each of these concepts refers to a type of artifact that needs to be designed, maintained, and evolved. This process domain knowledge needs to be represented in the environment, too.

A service-providing system in our domain is built (or evolves) by specific actions such as Create Service, Install Capability, and so on. These process operations can be considered as services themselves, where the user is the software designer/developer/maintainer rather than the usual service customer. The ability to rapidly create new services and alter the enterprise systems to provide the services is critical and therefore comprise important (meta-)services in themselves.

Thus, we think that work on general models of software process should be specialized to specific domains.

Summary

By exploring some of the manifestations of domain-specificity in our domain of IN systems, we have found some representation concepts that could have parallels in other domains.

Note how some general SE principles were instantiated but restricted to impose constraints that help gain control over the domain.

> Domain-specific building blocks are like reusable components that result from domain analysis of the services and service-providing systems in the domain. However, they play a stronger role in constraining the design.
>
> Domain-specific levels based on design decisions are similar to levels of abstraction in software engineering but there is a fixed number of levels. We do not do an analysis/design to find out how many levels a system will have.
>
> We study systems in the domain and then design a fixed set of appropriate levels.
>
> Enterprise domain knowledge is similar to knowledge represented in generic environments. However, this knowledge is built into the environment (at the meta-level) to become part of a domain-specific framework.
>
> Process domain knowledge specializes the vocabulary and tools of the software process, so that domain experts have a more direct understanding of the process.

We close by mentioning a couple of issues that could be discussed at the workshop:

How can we build on general-purpose/vanilla methods and tools? Some fairly well-understood vanilla notions of behavior, function, structure, etc. are converging in AI. Similarly, some forms of object models, dataflow diagrams, state-transition, etc. from the CASE world are becoming fairly standard. We need to understand now to systematically construct domain-specific structures on top of (or next to) these.

Are there some common subdomains whose subject matter knowledge and design knowledge would be useful across several different domains? Is anybody working on representation and reasoning frameworks for important domains and packaging them to be shared across domains?

What is a "good" high-level design? For example, suppose high-level design includes assigning responsibilities to agents and assigning ownership of resources to agents. Then a "good" design might be one in which all agents who have responsibility for an action own the resources needed to carry out the action. However, this might be too restrictive; a suitable design might be one in which an agent responsible for an action either owns the needed resources or has access to an agent who does. This needs to be developed, and a framework for expressing designs is needed. (If there is already some work on this, we would like to become aware of it.)

References

Greenspan, S., M. Feblowitz, C. Shekaran, and J. Tremlett, 1991. Addressing Requirements Issues Within a Conceptual Modeling Environment. In Proceedings of the International Workshop on Software Specification and Design.

IEEE Communications Magazine, December, 1988. Issue Featuring Building the Intelligent Network 26(12).

IEEE Communications Magazine, February 1992. Issue Featuring Intelligent Networks 31(2).

Greenspan, S.1991. Analysis and Design of Composite Service Systems. In Proceedings of AAAI Symposium on Composite System Design.

Arango, G.; Prieto-Diaz, R., 1991. Introduction and Overview: Domain Analysis Concepts and Research Directions. In Prieto-Diaz, R., and Arango, G. (eds), *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

Duran, J.; Visser, J., 1992. International Standards for Intelligent Networks. in [IEEE Comm., Feb. 1992].

Reubenstein, H., 1990. Automated Acquisition of Evolving Informal Descriptions. Ph. D. diss., M.I.T. Technical Report 1205.

Interactive Specification Acquisition via Scenarios: A Proposal

Robert J. Hall
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, New Jersey 07974-0636
hall@allegra.att.com

Abstract

Some reactive systems are most naturally specified by giving large collections of behavior scenarios. These collections not only specify the behavior of the system, but also provide good test suites for validating the implemented system. Due to the complexity of the systems and the number of scenarios, however, it appears that automated assistance is necessary to make this software development process workable. ISAT is a proposed interactive system for supporting the acquisition and maintenance of a formal system specification from scenarios, as well as automatic synthesis of control code and automated test generation. This paper discusses the background, motivation, proposed functions, and implementation status of ISAT.

Note: This work is still in its early stages; comments, criticisms, and literature pointers are not only welcomed, but actively sought.

Background and Motivation

Some reactive systems, such as telephone switches and other control systems, seem to be most naturally specified informally by giving a set of behavior scenarios, consisting of interleaved sequences of applied stimuli and verified system responses. Here is such a scenario from the domain of telephone switches:

> **Assumptions:** X, Y, and Z are idle stations.
>
> **Stimulus:** Y activates Call Forwarding to Z.
> **Response:** Y receives a confirmation tone.
>
> **Stimulus:** Place a call from X to Y.
> **Response:** Y receives a redirect notification.
> **Response:** Z rings.
>
> **Stimulus:** Z answers the call.
> **Response:** X and Z are connected.

A comprehensive collection of these scenarios forms both an informal system specification and a suite of system tests. There are, however, several major problems with incorporating this specification and testing technique into a software development process:

- *Too many scenarios.* For systems as complex as modern phone switches, for example, there are too many scenarios (typically many thousands) for the entire suite to be manually executed even once per software release (of which there may be many per year). Furthermore, if bugs are found during an execution of the scenarios, there is no time to go back and revalidate each bug fix.
- *Ambiguity.* Informal English descriptions can be ambiguous. For example, the scenario above does not specify the technique that Y must use to activate Call Forwarding, but some such techniques may not result in a confirmation tone. Thus, the outcome of the test execution can be dependent on how the ambiguity is resolved.
- *Inaccuracy.* Informal English descriptions can be incorrect. For example, the default administration of stations dictates that they do not receive redirect notifications unless this feature is explicitly activated. Thus, the scenario above will fail, unless the tester inserts the missing administration step.
- *Consistency Maintenance.* It is difficult for humans to maintain the mutual consistency of these scenario sets as the system evolves over time, and the developers come and go. For example, an early scenario may specify that a call to a busy station is denied, with the caller receiving a busy signal; subsequently, the system may be changed so that a later scenario specifies that a call to a busy station is redirected to an automated answering feature. This change causes the first scenario to fail in system test.
- *Testing Pragmatics.* In running a batch of system tests sequentially, it is crucial that one test leave the stations in a known "default" state, so that subsequent tests' initial assumptions are satisfied. The scenario above violates this felicity condition by leaving Call Forwarding activated and leaving X, Y, and Z offhook. Thus, a batch test run can fail, even

though the software is really correct, simply because of the ordering of the tests.

A First Try: KITSS

In the KITSS project (Nonnenmann and Eddy, 1992),[1] the goal is to ameliorate these problems by translating from English into a formal test script language which can be automatically executed on a system test harness. KITSS operates in the domain of telephone switch software, using a knowledge-based domain reasoner that both assists in the translation and audits the scripts for consistency with a domain model. KITSS has the potential to help with all of the problems noted above: the translation process disambiguates the input, using sophisticated reference resolution and pragmatic reasoning, as well as a library of domain plans. Next, the auditing phase maintains consistency with the domain model and detects inconsistencies resulting from inaccurate scenarios. Finally, a planner uses domain knowledge to repair incomplete plans and to restore the system state at the ends of scenarios.

While it is beyond the scope of this paper to analyze all of the successes and failures of the KITSS project, I believe there are several lessons of the project with direct impact on this proposal.

The reasoner's model cannot be static. One basic assumption of KITSS is that there is a highly complete and virtually *static* domain model that can be built once and changed only very slowly. If this were true, then the effort of building this model could be amortized over all applications of the system. Unfortunately, change seems to be the rule rather than the exception. In one case study of only the knowledge required to support the natural language semantics module, I calculated that I had to add (or change) roughly one *knowledge unit*[2] for every five sentences processed successfully. This was measured in adding the knowledge required to allow the system to correctly translate all sentences of 38 scenarios (roughly 400 sentences). Moreover, the frequency of knowledge addition was not "converging" as the test coverage grew. The simple reason is that broader coverage means new things to talk about and new ways of talking about things. While this is hardly a definitive study, it is nevertheless suggestive that the domain model will constantly undergo evolution, rather than remaining fixed.

Experience with other KITSS system modules, such as the automated reasoner and the natural language parser, indicates this phenomenon applies to them as well. Each time a system release includes a new feature, the reasoner's domain model must be extensively updated to allow for it. Even if no new feature is added in a release, it is typical that some aspect of the specified behavior is either changed or simply better defined. Commonly, unanticipated *feature interactions* need to be defined or repaired; for example, it may be necessary to change what happens if a Priority Call is placed to a station with Call Forwarding active, since such calls are not treated as normal calls. Of all KITSS modules, only the natural language parser (Jones and Eisner, 1992) has addressed the issue of automating the acquisition process.

The natural language semantics problem is too hard. At the start of the KITSS project, it was believed that the natural language used in writing the scripts was constrained enough to make possible automatic understanding. While this seems to be true for the *syntactic* aspects of the English used (Jones and Eisner, 1992), it appears that the *semantic* aspects are wildly unconstrained, with sentence styles varying from simple and action-centered to elliptic, imprecise, inaccurate, subjunctive, and even metaphorical. An example is

> Station B2 wants to talk privately with Station B1, so presses the Consult button.

This is elliptic, in that the second half of the compound sentence leaves out who is pressing the button. It is metaphorical in that stations cannot really have desires, and cannot really talk. To fully handle phenomena such as metaphor and ellipsis, a system must have a great deal of pragmatic, common sense knowledge. It is well known that the problem of common sense knowledge is extremely difficult. A result of this observation is that we cannot depend on perfection in the natural language component, so an interactive interface is required that makes it possible for the user to examine each translation and repair it as necessary.

The benefits of imperfect natural language processing may not justify the knowledge and processing costs. In another informal study, I used KITSS to translate 14 test cases. I did this in two ways: first, by having the natural language component try to translate the sentence and only repairing those sentences inaccurately or not translated; and second, by simply manually typing the translation essentially directly into the logical language used by the domain reasoner. The session which used the natural language component required 47 minutes, while the session without only required 49 minutes. The key reasons for this, I believe, are that (1) the translation is usually extremely easy to find for someone familiar with the logical language, and (2) the processing time in the domain reasoner was large enough that there was plenty of time for the human reasoner to think about the paraphrase in parallel with this processing.

[1] KITSS was reported on in last year's Workshop (Nonnenmann and Eddy, 1991) as well.

[2] A "knowledge unit" is a qualitative unit of effort defined with respect to the knowledge formalisms employed in the KITSS system. Its key properties are that it must be added manually to the system by a relative expert in the domain model, and each knowledge unit is of roughly the same complexity to add (as measured by the time to discover and add it).

A New Approach: ISAT

The subject of this proposal is a new tool called ISAT, for Interactive Specification Acquisition Tool. The first point of departure with KITSS is to acknowledge the model acquisition and maintenance problem as the overriding problem. This has impact throughout the tool's design, starting with a different role for the user. Whereas in KITSS the user's task is knowledge-assisted translation of scenarios, given a static system and domain model, the ISAT user's task is to synthesize a system model (specification) which is consistent with the scenarios. Translation of the scenarios into automated test scripts is a by-product of this process, rather than the primary goal. Thus, whereas the users of KITSS are system testers, the users of ISAT are the developers and designers of the system. Of course, the testers still benefit from fully automated and maintainable test scripts.

Note that there is a subtle difference between KITSS's domain model and ISAT's system model. The domain model in KITSS has evolved into a collection of constraints, plans, and inference rules. It is not, however, a predictive model of the switch, as this would require *completeness*. Such completeness is impossible to attain in a system with a fixed domain model. Thus, KITSS is capable only of checking certain aspects of scenario consistency, and uses plan recognition to fill in missing scenario steps.

By contrast, ISAT's system model *is* assumed to be predictive. It must be complete enough to predict all observations in all scenarios. Whenever there is an unpredicted observation or an inconsistency, the user must fix either the scenario or the model. By designing ISAT for maintainability, however, this is acceptable. Note that I will use the terms "specification" and "system model" interchangeably throughout to denote a predictive model of the stimulus/response behavior of the target software.

One might wonder why it should be possible to acquire such a specification from the user, since traditionally software specifications have been extremely difficult to produce. There are two answers to this question. First, the goal is to acquire a behavioral, input/output specification of the system similar to what one might find in a user's manual for the target system. Since, for example, there are those who claim to understand how to use their phones, we might expect this level of specification to be much simpler than a full specification of the switch software itself. A full specification would include far more detail than is normally seen by a user, such as constraints imposed by hardware resources. Second, in the ISAT project I am not requiring a complete and accurate specification up front. Instead, the specification is fundamentally an evolving entity which undergoes continuous, but controlled, change. By designing for maintainability, and supporting automatic code synthesis (see below), ISAT sidesteps the difficulties of full specification.

In view of the observations above about natural language processing, ISAT will not accept English input; rather, it will accept formal input only. Thus, each scenario must be manually transcribed into a formal stimulus/response language. Furthermore, the system model itself will be expressed in a formal rule language with a clear semantics.

I believe that this problem redefinition, even though it places a larger burden on the user, allows much more leverage on the testing and maintenance problems. The next section will discuss in detail the benefits which I believe should accrue from this change. Broadly speaking, ISAT (like KITSS) is an *apprentice* system, i.e. one which assists engineers in doing a task by automating the routine subtasks and tracking as many details as possible. Examples of this paradigm in the literature abound: the LEAP system (Mitchell, et al, 1985) was an apprentice VLSI design assistant, and the MIT Programmer's Apprentice Project (Rich and Waters, 1990) supported research on many different apprentice systems, such as KBEmacs, the Design Apprentice, and the Requirements Apprentice (Reubenstein, 1990).

Proposed Tool Functions

Through an extended interactive dialog, augmented by batch-mode processing of various subtasks, the system supports the user in constructing a predictive model of target system behavior that is complete enough to predict every response in every test scenario. With such a model, several important software development tasks can be carried out. The primary functions of the proposed ISAT apprentice system are given here and discussed in more detail below.

- *Scenario checking.* Verify that each response in a given scenario is predicted by the model, given the assumptions and stimuli in the scenario.
- *Model Maintenance.* Control and analyze a user's changes to the system model, performing impact analysis and regression testing to ensure that such changes are consistent with all known scenarios. Provide diagnostics and explanations when conflicts arise.
- *Automatic Programming.* At any time when the system model is known to be consistent with the scenarios, compile the model into an efficiently executable control module for the target system.
- *Generation of Automated Test Scripts.* Put out scripts in the low-level executable form necessary for execution on a test harness. This includes filling in missing steps necessary to leaving the system in the default state, error recovery, etc. (This is essentially the KITSS task.)
- *Test Suite Enhancement.* Fill in individual scenarios with additional response verifications that were left out of the input scripts, based on the predictions of the model. Possibly suggest additional scenarios

for testing known gaps in scenario coverage, such as model rules that are never fired or state variables that never change.

Scenario Checking

The fundamental mode of interaction in acquiring the model is for the user to present a scenario to the system, which the system then "simulates" using its current system model (represented in a simple pattern-action rule form based on a simple notion of state). The system then informs the user whether the behavior specified by the scenario is successfully predicted by the system's model. Exceptional conditions are raised when any of the following conditions arise:

- a response specified by the user *contradicts* some deduced consequence of the system's model,
- a response specified by the user, though not contradictory, fails to be predicted by the system's model (indicating possible incompleteness of the model),
- a stimulus applied by the user is deduced to be illegal in the system's current state
- the system model reaches an inconsistent internal state (which may not be observable as a system response in the scenario).

Whenever such an exception arises, it could be due either to an incomplete or inaccurate system model held by the system, or to an incomplete or inaccurate scenario presented by the user. Thus, the first important automated facility of the ISAT system is the ability to *fully explain* any deduced state condition. This explanation is presented in terms of the pattern-action rules constituting the system's model and the scenario stimuli and configurational information given by the user. The user can then use this explanation to isolate the difficulty, resulting in either changing the scenario definition or fixing the model.

Note that this explanation facility hinges on a key property of reactive system control software: each event results in a relatively small number of internal state changes. This allows us to construct a fully explained and complete execution trace of the system model on a given scenario input. This property does not hold true of other types of software, such as data processing software, compilers, etc. They typically have enormous traces, involving millions of internal states. It is practically impossible to build and query a complete trace of such a system.

Another function potentially performed by the checker is to compare the final state of a scenario with the assumed default initial state of all scenarios. ISAT can then point out when the state is left in a non-default state, and even assist in planning some steps for correcting it.

Model Maintenance

If the difficulty lies in the system's model, the user must decide how to repair the model. Usually, for any given model repair, the biggest difficulty lies in understanding how the proposed change will effect the correctness of the system on *other* scenarios. That is, does this fix break anything that worked before?

In ISAT, this is not a problem because of our insistence on complete explainability. In principle, each scenario can be re-checked; any that no longer complete successfully provide full explanations of why they fail, allowing the user to quickly locate the unintended interactions. In practice, we can speed up this process by orders of magnitude for small model changes by examining the justification structures built up in originally checking them; a small model change will not effect many scenarios, so this checking can quickly determine that the original structure still applies (this is analogous to the difference between deriving a proof and checking an existing proof). In my experiments with the current prototype of ISAT, this simplified impact analysis is roughly 40 times faster (for localized model changes) than a full recheck of all scenarios would be.

The above technique applies to changes in model rules; I anticipate there will be analogous protocols for dealing with other types of model changes, such as changing the types of model functions, adding and deleting state variables, etc.

Automatic Programming

Since the system model is predictive, it can serve as a controller for the target system, assuming the hardware and low-level primitives support the stimulus and response primitives directly. Specifically, I assume that the system substrate can be coded to supply typed interrupts when sensors indicate the presence of real stimuli (such as when a button is pressed, or a phone goes offhook). I also assume that the substrate supports actuators for each of the observable signals deduced by the system model (such as a status lamp lighting or a tone being emitted). Given this substrate, which may or may not exist in current day designs, we can synthesize a controller by essentially treating the sensor interrupts as stimulus statements in a scenario. Then, when the system model computes the next observable system state, the changed observables are sent as updates to the actuators.

The only difficulty with this direct approach is the efficiency of the controller: even if there are no hard real time constraints (which there may be), large systems like phone switches must handle many interrupts per second to be usable. Fortunately, I believe it should be quite possible to compile the system's model into an extremely efficient program. The run-time system need not track rule justifications, and rule chains can be precomputed at compile time. Thus, the event handlers for the system at run-time needn't do term matching, justification construction, or consistency computation.

Note that there are several benefits to this automatic programming approach.

- *Initial Coding is fast.* This is because the control

part is generated from the system model automatically. Because of the extremely simple model of computation in the system and the limited domain, I hypothesize that this automatic programming problem can be fully automated.

- *Subsequent releases are relatively painless.* In many domains, new system releases tend to involve mostly changes to the high-level control, rather than the sensor-actuator substrate. Thus, future releases can be produced by first getting them correctly reflected in the ISAT system model and then automatically regenerating the controller, leaving the system substrate the same.
- *Bug fixing turnaround is fast.* Another benefit lies in debugging the actual system. If a user calls up with a bug report, it can be simulated in ISAT's model, to see if it is replicated there. If it is not, then the bug is localized to the sensor-actuator substrate. More likely, however, is that the bug is manifested in the model, where the full explanation facilities of ISAT allow quick localization, fixing, and regression checking. This can potentially lead to extremely rapid turnaround for bug fixing. Note that quick bug localization based on querying scenario traces depends on the special properties assumed of this class of reactive control systems.

Even though I believe the automatic programming task in this domain is tractable, challenges remain. For example, compiling arbitrary rule patterns into efficient code is still challenging. For example, when a rule's condition contains the pattern `(connected X ?y)`, the naive compilation performs a linear search among all stations to see which are connected to B1. It would be desirable to compile this into a hash-table based representation of the set of connected stations to X. There is much existing research on this and related compilation problems, however.

Generation of Automated Scripts

The challenge here is to translate from the high-level stimulus and response statements appearing in the scenarios into the particular low-level language used by the test harness. This will involve two steps. First, each high-level step, such as "Activate Call Forwarding from Station X to Station Y" must have one or more *compound action methods* defined telling the system how to translate it into a sequence of primitive stimuli understood by the system. I believe this is properly part of the system model acquired from the user. The final step is to do a more-or-less standard compilation step from the event primitives in the model into the language of the test harness. The event primitives should be designed to make this relatively easy.

Note that the checking phase of ISAT is presumed to have already made sure that each scenario leaves the system in a known default state.

Test Suite Enhancement

There are two different types of enhancement that ISAT can easily perform: first, each single scenario can be "filled out" with missing observations, to increase confidence in the proper working of the switch. It can do this based on the predictions of its model. For example, it might insert checks for dial tone after each off-hook operation. Of course, we must be careful not to bog down the scenarios in endless checking of details, since there are so many to execute on the test harness.

The second type of enhancement is to the coverage of the suite as a whole. If a model rule is never fired in any scenario, this probably indicates that one or more scenarios should be created to exercise it. (Otherwise, why would the user put it in?) Similarly, if a certain type of state variable either always has the same value or is never determined in all scenarios, scenarios should be created to see if it is possible to cause a change. At the very least, these types of conditions can be brought to the user's attention. More ambitiously, the system can do goal-based planning to try to bring about the conditions necessary to firing the rule or changing the variable's value.

Implementation Strategies

To date, I have implemented an initial prototype demonstrating some of ISAT's capabilities. In particular, the system can check scenarios and perform impact analysis for individual rule changes. It can also emit low-level streams of stimulus/response statements as the first step in producing automated scripts. I have used ISAT to build several different models of different combinations of telephone switch features. The most complex is a model that layers Priority Calling and Call Forwarding on top of Plain Old Telephone Service for multiple call-appearance phone stations. Since this model acquisition was done in parallel with ISAT implementation, it is premature to attempt to gauge how well I was supported in this by ISAT. I have used the model to successfully check 17 scenarios taken from actual pre-existing AT&T system test documents.

The formalism I've adopted is based on simple pattern-action rules used to form partial descriptions of state transitions.

```
   if (CALL-STATE (CA ?X ?N)) = :IDLE, and
      (SELECTED-CA ?X)        = (CA ?X ?N), and
      (HOOK! ?X :OFF)         = :TRUE,
 then (CALL-STATE (CA ?X ?N)) = :PRE-DIALING
```

This rule says that if call appearance number ?N at station ?X is idle and is the selected appearance of ?X and ?X goes offhook, then in the next system state the CALL-STATE of that appearance is ":PRE-DIALING". There are two types of model rules: state change rules, like the above, are used in a "forward" manner; that is, they are executed to quiescence after each stimulus event is applied. Demand rules, like

```
if (CALL-STATE (CA ?X ?N)) = :DENIED, and
   (SELECTED-CA ?X)        = (CA ?X ?N), and
   (ONHOOK? ?X)            = :FALSE
then (RECEIVED-TONE ?X)    = :BUSY-TONE
```

are only used when a response event asks about one or more of the predicates in the rule's conclusion. Thus, the above rule will only be used when the scenario executes an observation of the received-tone of some station. Demand rules are necessary so that the system need not forward chain to deduce a large number of observations that won't be used in a given state. For example, there are quadratically many potential connections between stations, but by making the observable connected predicate only deduced on demand, the model can execute in linear time.

Note that, unlike state rules, the demand rules do not chain arbitrarily. They are used only one level deep. This makes the system's efficiency much more predictable, and I have found it no significant restriction in expressive power.

ISAT deals with the classical AI frame problem (how to consistently carry forward unchanged facts when a small aspect of the state changes) by distinguishing different declared ontological statuses of terms. Any term consisting of an application of a function declared :PERSISTENT by the user has its old value brought forward, unless an explicit contradiction is deduced by a rule firing. Any non-persistent terms must be rededuced in each state. Most such terms are deduced by demand rules, though, so they do not incur a large unnecessary cost.

Because of the extremely simple formalism and semantics, the current prototype is able to support several useful explanation functions. Of course, it can answer (WHY? <fact> <state>) by simply giving the explicitly maintained justification in terms of rule applications, external inputs, etc. Another extremely useful capability is the ability to answer (WHY-NOT? <fact> <state>). Obviously, in its most general form, this is too open-ended to be meaningful; but in the context of ISAT this is interpreted to mean the following. First, tell me any contradictory facts (optionally explaining them); then, tell me all the rules that could have deduced the fact and tell me why they didn't fire (by telling me which of their conditions are not satisfied). This has been very useful in building the models I have already built. An analogous facility is (METHODS? <action statement> <state>), which gives a description of which compound action methods apply in the state for the given compound action application, and which fail to apply and why.

Summary

The proposed tool, ISAT, is a software development tool environment for reactive control systems, such as telephone switch software. It is motivated by, and builds on lessons learned from, the KITSS project. By redefining the problem and meeting the model acquisition problem head-on, I believe many major benefits are achievable, such as specification acquisition and maintenance, automatic synthesis of control systems, test enhancement, and automated script compilation. An initial prototype of ISAT is currently under construction, with several of the main functions implemented. It has been tested on several scenarios from a "real world" application.

Acknowledgements

This proposal has grown out of work on the KITSS project at AT&T Bell Labs. I have learned a great deal from this project and the people involved with it: Van Kelly, Mark Jones, John Eddy, and Uwe Nonnenmann. Some of the ideas and opinions expressed in this paper are clearly derived from the conceptual foundation of KITSS. However, I make no claim as to whether the individual project members (except me) agree with the specific opinions expressed here.

References

Jones, M.A.; and Eisner, J.M. 1992. A probabilistic parser applied to software testing documents. In Proceedings of the Tenth National Conference on Artificial Intelligence. Cambridge, MA: MIT Press.

Mitchell, T.; Mahadevan, S.; and Steinberg, L. 1985. LEAP: A learning apprentice for VLSI Design, In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, vol 1, pp 573–580. Los Altos, CA: Morgan Kaufmann.

Nonnenmann, U.; and Eddy, J.K. 1992. KITSS – A functional software testing system using a hybrid domain model. In Proceedings of the Eighth IEEE Conference on Artificial Intelligence Applications. Montery, CA: IEEE.

Nonnenmann, U.; and Eddy, J.K. 1991. KITSS – Toward software design and testing integration. In Proceedings of the AAAI-91 workshop: Automating Software Design: Interactive Design. AAAI.

Reubenstein, H. 1990. *Automated Acquisition of Evolving Informal Descriptions*, Technical Report, AI-TR-1205. M.I.T. Artificial Intelligence Laboratory.

Rich, C.; and Waters, R. 1990. *The Programmer's Apprentice*, New York, NY: ACM Press.

DISTRIBUTED INTELLIGENT CONTROL AND MANAGEMENT (DICAM) APPLICATIONS AND SUPPORT FOR SEMI-AUTOMATED DEVELOPMENT[1]

Frederick Hayes-Roth, Lee D. Erman, Allan Terry,
Teknowledge Federal Systems,
Cimflex Teknowledge Corp.

& Barbara Hayes-Roth
Knowledge Systems Laboratory
Stanford University

Introduction

We have recently begun a 4-year effort to develop a new technology foundation and associated methodology for the rapid development of high-performance intelligent controllers. Our objective in this work is to enable system developers to create effective real-time systems for control of multiple, coordinated entities in much less time than is currently required. Our technical strategy for achieving this objective is like that in other domain-specific software efforts: analyze the domain and task underlying effective performance, construct parametric or model-based generic components and overall solutions to the task, and provide excellent means for specifying, selecting, tailoring or automatically generating the solution elements particularly appropriate for the problem at hand.

For intelligent control tasks, we believe that the domain-specific software approach holds the promise of providing great leverage on the software development task whether software generation is manual, automated, or semi-automated. In our view, complex and mission critical systems generally must have a human analyst in the loop both to specify desired behavior and to validate tested designs. Until this process is made extremely regular and routine, the human will necessarily be involved in nearly every step of the software development process as well. Given the lack of regularity and proven automatic generation means, the human's ability to validate overall designs requires insight into and hands-on experience with the details of the software design and generation. Nevertheless, we believe that significant progress on the "time to market" for such systems requires much of the same supporting infrastructure, regardless of the extent to which productivity enhancements are achieved through automation or merely improved methodology. This position is similar to that held by experts in many fields who state that one should not automate poorly characterized, highly variable processes. First, we must attempt to regularize the process, support it with an effective and efficient methodology, and then automate those portions of the process which give the greatest return on investment.

In this paper, we first present our specific domain focus, briefly describe the methodology and environment we are developing to provide a more regular approach to software development, and then later describe the issues this raises for the research community and this specific workshop.

Project Objectives and General Approach

Our project aims to develop a new technology foundation and associated methodology for the rapid development of high-performance intelligent controllers. These controllers will be employed in distributed intelligent control and management (DICAM) applications. Examples of such applications include intelligent highway systems, military command and control systems, and factory floor control systems. Our near-term domain of application is vehicle management systems, where one or more controllers may be employed to control a single vehicle, and these composite controller/vehicles are further aggregated and organized into higher-levels of control and capability. In a military context, for example, a single controller may be used for each subsystem within a tank, each tank system may be controlled by collectively organizing its subsystems, the overall tank may be controlled by another controller that coordinates the tank system controllers, several tanks may combine to form a platoon with its own control level, one or more platoons may form a battalion, and so on.

Our research project is one of several sponsored by DARPA (the Defense Advanced Research Projects Agency of the US Defense Department) and the US Army to advance the technology for domain-specific software architecture (DSSA). Our project for the Army address the specific vehicle management task of a howitzer, a tank-like vehicle that aims at more distant targets. The project has four principal focus elements. First, we are formulating a *reference architecture* for intelligent control. Second, we are supporting the construction of applications in a *development workspace* in which system requirements are ultimately satisfied by choosing design components that specialize and particularize components of the generic reference architecture. Many of the specialized modules and particular data used to instantiate a design are taken from a *repository*. The entire development process is supported by a rich array of *development tools*, which incorporate numerous techniques from both software engineering (e.g., control law specifiers, code generators, protocols, compilers, debuggers) and knowledge engineering (e.g., domain modeler, requirements manager, and various knowledge-based design assistants).

[1] This work reported here has been supported in part by DARPA and the US Army ARDEC through contract number DAAA21-92-C-0028. The opinions expressed here are those of the authors, not the sponsors.

The DICAM Framework and Supporting Technology

We are developing DICAM simultaneously as a "model" or *framework* for understanding control problems and as an *architecture* and related *environment* for building controllers. There are many reasons why we seek to formulate such a unifying framework. Foremost among these is our belief that the difficult, time-consuming and often unsatisfactory process of controller development would benefit from a more "standard" but flexible approach. Our DICAM framework provides a generic but customizable model of controllers that seems to unify a variety of views and experiences in the control, software and knowledge engineering disciplines.

DICAM is closely related to the NASA/NBS reference model for telerobot control systems (NASREM) [Albus, McCain, & Lumia, 1989]. The reference architecture includes two principal components in any distributed intelligent control and management application. First, an *information base and world model* (IB/WM) is a "conceptually centralized" database/knowledge base that represents the state of the world. The second principal component of the DICAM reference architecture is a collection of semi-autonomous interconnected controllers. These controllers are differentiated in terms of the scope of behavior they address, the resources they control, and the time frame spanned by their decisions.

Each controller is actually divided into two separate but interrelated components called the *domain controller (DC)* and the *meta-controller (MC)*. The DC contains several modular functions and prescribes how they interact using dataflow conventions. The functions include sensing, input filtering, situation assessment, planning, plan assumption analysis, execution and effector activation. Each controller has its own local world module which is a cached view of the global state represented by the IB/WM. Several messages flow into and out of the DC. The inputs include messages received from a superior controller specifying goals for the controller, messages from sibling controllers at the same level (such as another vehicle in the same group), and messages from subordinates, typically reporting on the outcomes of their efforts. Outputs include subgoals assigned to subordinates for delegated execution as well as messages to siblings, for example, to report on current plan execution objectives or status or to request operating resources.

Although this general DC structure has proved effective in applications such as the Pilots Associate [Smith & Broadwell, 1989; Lark *et al.*, 1990] and robotics [Becker, 1989], dataflow programs in general exploit only weak knowledge about when to execute functions. The general rule is to compute any function when all of its inputs are available. However, there are often too many possible instantiations to execute all simultaneously, or even with a small delay. Thus, in situations where more knowledge is required to achieve excellent results with scarce resources, a metal-level of control is required [Garvey & Hayes-Roth, 1989; Hayes-Roth, 1985; Hayes-Roth, 1990]. Our meta-controller is based on the knowledge-based scheduler of the BB1 blackboard system. This controller utilizes three basic functions to determine on a cyclical basis which pending action is best to execute next: an *agenda manager* to store and evaluate pending actions; a *scheduler* to determine the next action based on the degree of fit between goals of a control plan and actions pending on the agenda; and, finally, an *executor* gives control to the selected actions.

Our basic methodology for development of DICAM applications consists of a blackboard-like environment where the "blackboard" is a *development workspace* and the "knowledge sources" are system developers augmented by a wide variety of computer-based tools, including some expert systems that are capable of autonomous development activity. We are assembling an Application Development Support Environment (ADSE) for DICAM applications (see Figure 1) to provide these capabilities.

The development workspace contains a representation of the emerging system being developed incrementally over time. Its elements represent decisions or specifications linked into a "web" of mutually supporting decisions that both specify the system design and justify it. We have combined three lines of research in formulating this development workspace. First, we have drawn on the blackboard model and opportunistic reasoning [Erman *et al.*, 1980; Hayes-Roth & Hayes-Roth, 1979; B. Hayes-Roth *et al.*, 1986] as an organizing methodology for incremental design and development processes. Second, we have adopted the emphasis of the domain analysis and domain-specific architecture approach to software specification, reuse and rapid development [Prieto-Díaz & Arango, 1991]. Lastly, we have adopted and generalized the approach of module-oriented programming from our previous research on ABE [Erman, Lark & Hayes-Roth, 1988; F. Hayes-Roth *et al.*, 1989; F. Hayes-Roth *et al.*, 1991]. This includes the ideas of recursive modular composition, distributed control through message passing using ADTs, system construction through module composition, and system realization by deferred binding of processors to modules.

Specifically, the workspace provides a multi-faceted, multi-level representation of DICAM software applications. It provides means for describing the domain model, i.e., the general characteristics of the task and environment in which the vehicle or machinery will operate. The general domain model is then augmented with specialized information about the specific application being built, such as how many vehicles, the distances to be traveled, the specific threats and so forth that the application will address.

At a lower (more concrete) level, the workspace provides means for representing the functional components and the physical resources that make up the controlled system, and it describes how the functional components are composed and how they are implemented using specific processors, communication capabilities or other machinery.

In addition, the workspace provides means for representing the status of the software development process, including the history of activities and characteristics of the current overall development.

As is typical of blackboard systems, the workspace provides means of representing decisions and using state change to trigger the invocation of appropriate tools. Decisions in this workspace range from abstract characterizations of components such as requirements or

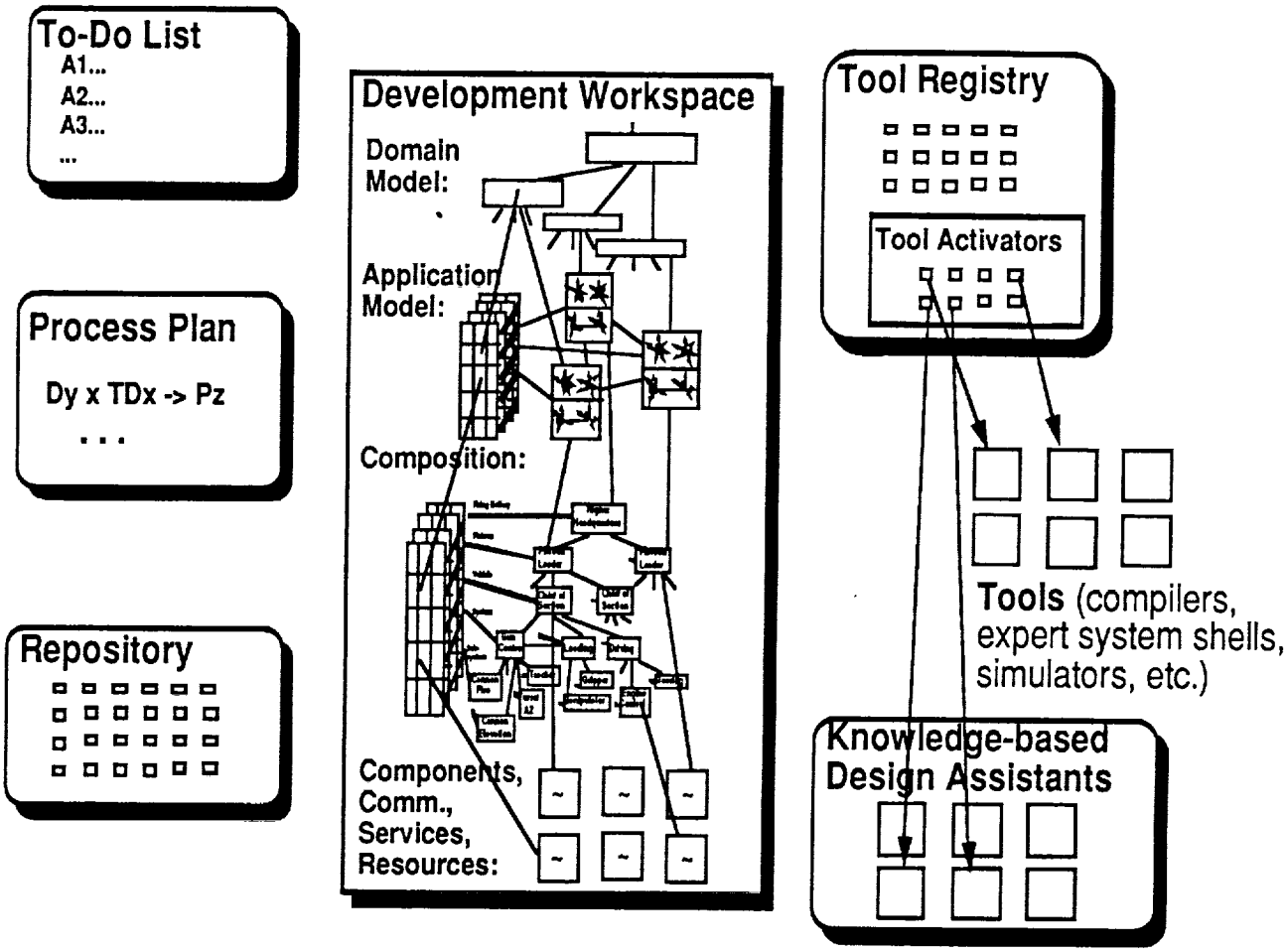


Figure 1. The Application Development Support Environment.

goals to particular specifications, including detailed functional characterization or specific software or hardware packages that realize the required capability. We have not yet settled upon final or formal representation sublanguages for each level, but are considering various alternatives that are being suggested in other groups' efforts to conceive potentially standardizable descriptions of modules and module interfaces (e.g., the DARPA module interconnect formalism, the DoD STARS repositories, etc.). Regardless of which specific formalism is used, the description of modules must include input/output datatypes, function characterization, implementation requirements, domain assumptions, and performance metrics. When making a design decision, the developer specifies some or all of these attributes along with his or her name and some rationale. As in all blackboard systems, decisions are changeable, and multiple competing decisions may coexist. Ultimately, those decisions that form the best coherent "web" win: these decisions constitute the overall system specification, from requirements to implementation, which particularizes the domain and application models.

Other features of the ADSE that we are developing and assembling include: A *To-Do List* keeps an agenda of pending tasks for the software developers. As with blackboard systems, actions are triggered when the state of the Workspace matches a pattern of interest. A *Process Plan* is supported that effectively maps patterns of interest found in the Development Workspace and the current To-Do List into proposed actions. The proposed actions (shown as Pz) in the figure might include any of the following: make a specific design decision; apply a particular tool to a particular design component with certain parameters; raise the priority on doing one pending task over others, etc. Our plan is to support a wide variety of SE methodologies by providing a general mechanism for representing and implementing corresponding process plans. A *Repository* of reusable components is provided that stores, classifies, and searches for previously used Development Workspace structures. Typically these include reusable domain models, application characteristics, generic function modules, specific implementation modules, and data to customize or particularize generic functions for specific application domains. A *Tool Registry* provides mechanisms for enrolling software development tools, describing their required inputs and associated outputs in terms of patterns that match characteristics found in the Workspace or Process Plan and, finally, providing *Tool Activators* that can automate or semi-automate invocation and application of tools. The tools consist of compilers, generators, simulators, expert system shells, etc. Lastly, the ADSE incorporates specialized tools called KBDAs that provide knowledge-based assistance in to the software development process. These tools can include, for example: requirements

Table 1. Aspects of the Development Methodology

Aspect	Elements
Opportunistic Design	Multiple levels and representations Abstract to particular characterizations Incremental decisions Linked decisions form design web Prescriptive process models permitted Humans and computer tools cooperate
Controller Architecture	Generic modules Flexible, tailorable controllers Schema of ADTs for IB/WM Message processing using ADTs and intermodule protocols Distribution Fractal control model
Information Base/ World Model	Shared data managed by IB/WM Conceptually centralized, single-copy, but allows physical partitioning Typically distributed Time response must satisfy requirements ranging from sub-millisecond to a few seconds Different levels of aggregation Different meta-types: data, propositions, rules, plans Temporally organized and continually renewing
KB Design Assistants	Mini-expert systems watch process state and advise user at key events Tool-use expert systems help humans apply development tools
Repository	Stores and uses partial matches to retrieve "components" at any level Components classified in taxonomy from generic to particular Domain-specific customizations available to particularize generics
Engineering Foci	Domain modeling Requirements engineering Knowledge engineering, about DICAM and DICAM development tools and methods Performance objectives, measurement and attainment
Component Characterization	Goals & Constraints Models: Behavior, timing, functionality Interfaces: Datatypes Module partners Conversation types Protocols Messages to other devices Resource/environment prerequisites

analyzers that suggest appropriate reusable components; redesign advisors that suggest ways to modify an existing design in light of a change in requirements or capabilities; and intelligent interfaces that set up and run complex tools to assist a developer in generating or analyzing some code.

To implement the ADSE we are using a number of "off-the-shelf" technologies. Chief among these are: ABE [F. Hayes-Roth *et al.*, 1991], as an integration environment for tools, a composition framework for modular, real-time applications, and a catalog and classification system for the reuse repository; BB1 [B. Hayes-Roth, 1985], as an incremental workspace, process model interpreter, and agenda manager; M.4, a commercial expert system shell, for building the KBDAs; and the Requirements Manager (RM), a DARPA-sponsored software product for collecting, managing, and evaluating application requirements and validating application designs against requirements [Fiksel & Hayes-Roth, 1990]. We are also evaluating many other commercial and research SE tools for use in the ADSE [cf. NIST ISEE Working Group, 1991].

Development Methodology

The overall approach we are taking to development is summarized in Table 1. The seven principal facets fall into three basic categories of methods. The controller architecture and information base/world model constitute the reference architecture for the domain of intelligent control. The repository, engineering foci, and component characterization concerns define our approach to domain-specific software engineering. The opportunistic design and

KB design assistants define our approach to defining a process of software development that can, at least, be semi-automated.

We are currently applying the methodology to demonstration problems chosen from defense applications. As an example, consider the task of achieving intelligent control of field artillery systems, such as mobile howitzers. Howitzers, like other military vehicles, are self-propelled, mobile vehicles with offensive guns. Their primary mission is ground-based artillery shelling of over-the-horizon targets. They are very similar to tanks, armored personal carriers, and helicopters in general information processing terms. Thus, all military vehicles of this sort share elements of the domain model, but differ increasingly as these models and the corresponding application model become detailed.

The general DICAM architecture is specialized for Army Vehicle Management Systems by the selection of levels: battalion, battery, platoon, vehicle (section), system, subsystem. Then it is further specialized for a particular howitzer, e.g. the "ABC Howitzer," by the selection of functional controllers and their relationships. Each group of ABC howitzers is headed by a Platoon Leader who reports to higher headquarters. The Chief of Section of each vehicle reports to the Platoon Leader and is responsible for the Gun Control, Loading, and Driving functions.

Following the domain-specific approach, after developing the generic domain model, the next task for system developers is to elaborate the application model. The task application model enumerates desired functionalities associated with each level of control. Several generic functions appear repeatedly across different control levels, such as tasking subordinates with subgoals or performing external and system status analyses as part of situation assessment. These functionalities are also common across the analogous components in other vehicles: tanks, missile launchers, infantry fighting vehicles, etc. Thus, there are two levels of functional similarity:

- across different components within a vehicle, and
- across the similar components in different vehicles.

To convert the informal task analysis into a more formal, explicit application model, the system developer selects from among generic functionalities in the reference architecture, specializing and customizing them for the particular needs of the target application. Then to construct an application system, the developer uses these refined specifications either to select components from the repository that can perform these functions or to drive automatic, semi-automated, or manual code generation.

Issues Raised

Our research raises many issues, some of which are highlighted here:

- Is our methodology (as described in Table 1) effective?
- Does our reference architecture provide enough structure to make specification practical and component software reusable?
- How can a critical mass of reusable components be created?
- How can modules be characterized?
- How can the languages used for characterizing modules be standardized?
- How can modules be designed so that they can be specialized or customized to new applications?
- Which tasks in the development process are most worthy of automated support?
- How can the space generated by a diversity of vehicles, environments and control objectives be structured to maximize the potential for reusability of specifications and solution components?

References

[1] Albus, J. S., McCain, H. G., and Lumia, R. "NASA/NBS standard reference model for telerobot control system architecture (NASREM)," National Bureau of Standards, Tech. Note 1235. 1989.

[2] Becker, J. M. "The generic control level: a unifying view," *Proc. ROBEXS '89*, Palo Alto, CA, 1989.

[3] Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, R. "The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty," *Computing Surveys* 12(2), June, 1980, pp. 213-253.

[4] Erman, L. D., Lark, J. S., and Hayes-Roth, F. "ABE: An environment for engineering intelligent systems," *IEEE Transactions on Software Engineering*, 14(12), December, 1988.

[5] Fiksel, J. and Hayes-Roth, F. "A requirements manager for concurrent engineering in printed circuit board design and production," *Proc. of the Second National Symposium on Concurrent Engineering*, Morgantown, WV, February, 1990.

[6] Garvey, A. and Hayes-Roth, B. "An empirical analysis of explicit vs. implicit control architectures," in Jagannathan, V. and Dodhiawala, R. T. (eds.), *Current Trends in Blackboard Systems*, Academic Press, 1989.

[7] Hayes-Roth, B. "Blackboard architecture for control," *Artificial Intelligence*, vol. 26, pp. 251-321, 1985. Reprinted in: Bond, A. and Gasser, L. (eds.), *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann Publishers, Inc., 1988.

[8] Hayes-Roth, B, "Architectural foundations for real-time performance in intelligent agents," *Real-Time Systems: The International Journal of Time-Critical Computing*, 2(1/2), 1990, pp. 99-125.

[9] Hayes-Roth, B. and Hayes-Roth, F. "A cognitive model of planning," *Cognitive Science*, 1979, 3, 275-310. Reprinted in A. Collins and E. E. Smith (eds.), *Readings in Cognitive Science: A Psychological and Artificial Intelligence Perspective*. Morgan Kaufmann, 1988; and in J. Allen, and J. Hendler, and A. Tate (eds.), *Readings in Planning*, Morgan Kaufmann, 1990.

[10] Hayes-Roth, B., Johnson, M.V., Garvey, A., and Hewett, M. "Applications of BB1 to arrangement-assembly tasks," *Journal of Artificial Intelligence in Engineering*, 1986.

[11] Hayes-Roth, F., Davidson, J.E., Erman, L.D. and Lark, J.S. "Frameworks for developing intelligent systems: The ABE systems engineering environment," *IEEE Expert*, June, 1991.

[12] Hayes-Roth, F., Erman, L. D., Fouse, S., Lark, J. S., and Davidson, J. "ABE: A cooperative operating

Model-Based Software Design*

Neil Iscoe, Zheng-Yang Liu, Guohui Feng, Britt Yenne, Larry Van Sickle, Michael Ballantyne
EDS Research, Austin Laboratory
1601 Rio Grande, Ste. 500
Austin, Texas 78701
iscoe@austin.eds.com

Abstract

Domain-specific knowledge is required to create specifications, generate code, and understand existing systems. Our approach to automating software design is based on instantiating an application domain model with industry-specific knowledge and then using that model to achieve the operational goals of specification elicitation and verification, reverse engineering, and code generation. Although many different specification models can be created from any particular domain model, each specification model is consistent and correct with respect to the domain model.

Introduction

Although empirical field studies (Curtis, et al., 1988) have shown that application domain knowledge is critical to the success of large projects, this knowledge is rarely stored in a form which facilitates its use in creating, maintaining and evolving software systems. Capturing and managing this knowledge is a prerequisite to automating software design.

Unfortunately, domain knowledge is implicitly embodied in application code rather than explicitly recorded and maintained in separate documents. Even when documents are maintained separately from the code, the knowledge is stored in voluminous natural language documents in an informal rather than a formal manner. Although problem-specific languages are designed to remedy this situation, domain-specific knowledge is still captured in an ad hoc instead of a systematic manner. Furthermore, these languages are generally not designed in such a way that the results can be generalized or even replicated.

We are attempting to capture the domain-specific knowledge about different industry areas as a set of application domain models. Application domain models are representations of relevant aspects of application domains that can be used to achieve specific software engineering operational goals. Operational goals are always implicit in the construction of a domain model and are essential to understanding the form and content of that model. Unlike generalized knowledge representation projects such as Cyc (Lenat, 1990) that attempt to provide a basis for modeling encyclopedic knowledge, domain modeling explicitly acknowledges the commonly held view (Amarel, 1968) that representations are designed for particular purposes. These purposes–the operational goals–inherently bias any particular solution and dictate the final form of the model.

Many different operational goals and modeling projects are being pursued within the field of domain modeling (Iscoe, et al., 1991). This paper begins with an overview of the domain modeling research at EDS and our corresponding operational goals. We explain our approach to automating software design as a paradigm which facilitates the creation of multiple-specification models from a domain model. Finally, we discuss a set of issues that we have encountered in achieving our goals.

Programming-in-the-Large

EDS produces large software systems for a variety of industries such as utilities, finance, health insurance, and so on. Associated with each industry area is a rich body of knowledge which is critical to specifying and implementing the proper software system. This knowledge includes legal, financial, technical, and other expertise which is acquired by personnel over a period of years. EDS is organized into strategic business units (SBUs) so that the organization's knowledge about a particular industry can be leveraged through reuse.

At the EDS Austin research laboratory, we are building a domain modeling system which is designed to achieve the following operational goals:

- Requirements & Specifications—Eliciting, verifying, and formalizing software requirements and specifications,
- Program Transformation/Generation—Transforming a specification into efficient executable code,
- Reverse Engineering—Identifying the semantics of existing code in terms of a partial specification.

The realization of these operational goals is consistent with our long-term plan for creating knowledge-based tools to support programming-in-the-large (Barstow, 1988). The domain modeling approach provides ample opportunities for creating an automated software development paradigm.

* An earlier version of this paper was presented at the Asilomar Workshop on Change of Representation and Problem Reformulation, April 1992.

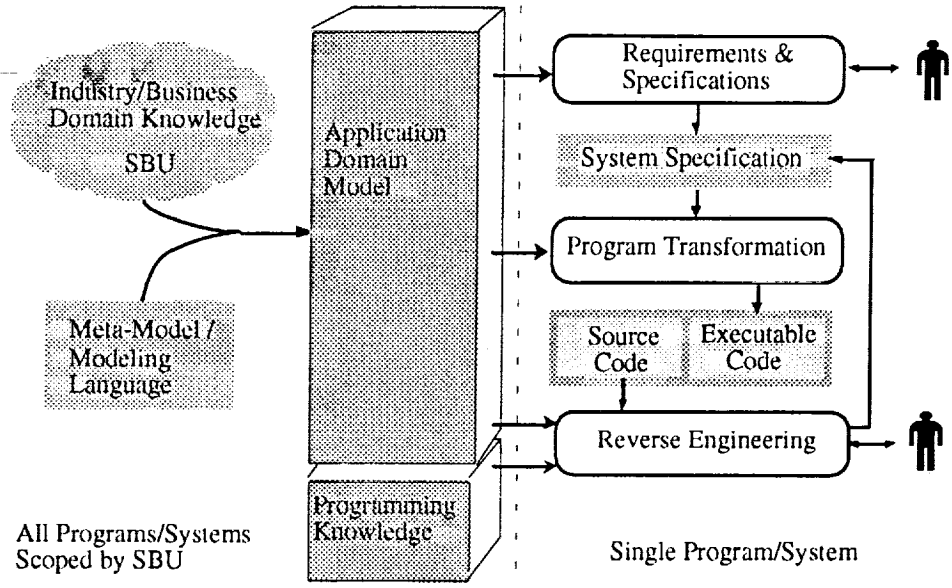


Figure 1 — Domain Modeling with Operational Goals

Figure 1 illustrates the context in which we operate. The industry knowledge for each SBU is instantiated into a domain model, which then serves as a source of knowledge for programs (the ovals) to achieve operational goals, such as reverse engineering source code or eliciting system specifications. The figure actually illustrates two different processes. The left side of figure 1 shows the process of domain model instantiation while the right side illustrates the domain model being used to produce a single specification. The *System Specification* (rectangle) illustrates a specification for a single specific system within an application domain. However, a multitude of system specifications can be created from a domain model.

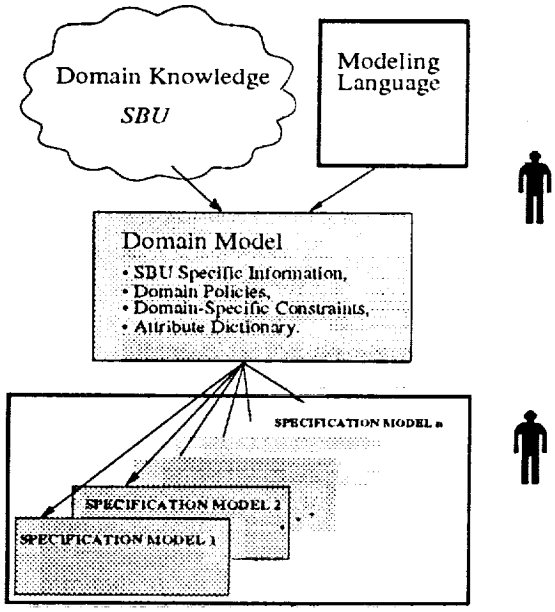


Figure 2 — Instantiating Specification Models

Figure 2 illustrates the two separate modeling tasks required by our approach. Domain experts interact with a system to represent their knowledge in terms of domain modeling constructs. Specification designers then use the system to build specification models which satisfy constraints in the domain model. In order to create a system specification, the application designer selects a set of relevant policies and constraints from the domain model that must be included and enforced in the specification model. The constraints include intra-attribute as well as inter-attribute relationships within and across classes relevant to the task at hand.

Because one of our goals is to generate executable code, we require that a particular specification model be consistent. A very large but finite number of specification models can be created which are consistent and correct with respect to a particular domain model.

Reverse Engineering

We are using reverse engineering to help instantiate both domain and specification models. Figure 1 illustrates how application domain knowledge and programming knowledge are used to extract partial specifications from source code. The box labeled "programming knowledge" currently represents knowledge of COBOL syntax, coding conventions, and program plans and structures (Van Sickle, 1992). This knowledge crosses all of the targeted application domains and is the basis of a separate code browser that operates independently of the operation shown in Figure 1.

We are also attempting to mechanically pre-instantiate domain models by using the data gathered from the applications of an EDS entity-relationship-based CASE tool that is used by SBUs for data modeling and code generation. By analyzing data models, we have access to tens of thousands of specific entities, relationships, and

constraints which have been used to specify programs and are useful for partially instantiating domain models.

Modeling Considerations

Models are inevitably abstractions of reality that capture information to achieve specific goals. A domain model determines the items of interest that exist in the world and sanctions the types of inferences allowed [Liu and Farley, 1990; Davis, 1991]. A model is the result of conscious decisions about what to describe and what to ignore. No model is complete or correct in the sense that it is applicable to all tasks.

Domain models in our system are structured to represent the type of information that is used within EDS SBUs to achieve our operational goals. Although EDS serves a wide range of industries, we are not attempting to model real-time or other application areas which diverge from standard business transaction processing. A general issue of interest in this research is the extent to which any particular representation/model can be mutated to hold different types of information for different tasks while still effectively achieving the original operational goals.

One requirement for our models is that they be consistent. Domain and specification model consistency is maintained by a specialized theorem prover. The theorem prover, *STR+VE*, is an upgraded version of the prover presented in (Bledsoe, 1980) for proofs of theorems in general inequalities. A TMS is being constructed to interface between the modeling system and the theorem prover.

Dynamic Knowledge Structure

The remainder of this paper presents one aspect of domain model representation and gives a glimpse of the relationship between specification and domain models and the organization of domain models.

While most would agree that hierarchical organizational strategies provide a reasonable way to structure knowledge within complex domains, the creation of a hierarchical structure, like any type of representational scheme, imposes a particular view of the world. Unfortunately, there is no particular view that is optimal for every application. Although the programs within a particular application share the same legal, physical, and economic constraints, the construction of any particular specification model depends upon a set of policy decisions that determine how cases are handled. Furthermore, *software in the large* systems are continually changing in such a manner that the concept of a static hierarchy is insufficient to capture the process of system evolution.

Consider software systems that manage the payment of health insurance claims. Although conceptually simple, these systems handle hundreds of thousands of different cases. One way to represent these cases is to enumerate the leaf nodes of the hierarchies created by the appropriate partitioning of attributes such as gender, age, family_status, previous_condition, employment, deductibles, copayments, prognosis, and so on. Unfortunately, the tree structure created by case expansion not only obscures relevant and interesting cases, but is also a monolithic structure. A paradox of object-oriented approaches is that well-adapted structures are not adaptable to new situations.

Because of the combinatorial explosion of the leaf nodes, it makes sense to handle the cases at as high a level as possible. Term subsumption systems such as CLASSIC (Borgida, et al., 1989) automate this process by determining the place in a hierarchy in which terms are subsumed. But subsumption systems assume a single structure in which all sub-models can belong. In the case of applications such as health insurance, individual modules may have different hierarchical structures and still maintain the integrity and constraint rules of the domain model.

Attribute Definitions

Attributes are normally considered as data values or slot fillers within a class or frame. However, the standard treatment of attributes as lists of data values with some underlying machine representation fails both to capture sufficient semantic information from the application domain and to state definitions with sufficient formality to allow semantics-related consistency checks.

Attributes are functions which define how a set of objects is mapped within a class. One type of attribute has a value set represented by a nominal scale which consists of a set of values, $\mathcal{V}(A) = \{C_1, \ldots C_n\}$.

One of the ways that the modeling process maps the world into a domain model is by creating categories in such a way that items to be categorized with respect to a particular attribute are as homogeneous as possible within a category and as heterogeneous as possible between categories. Examples of nominal scales abound and map cleanly to the notion of enumerated type as shown below:

```
(Colors
    :type     nominal_scale
    :values   (Red Yellow Green Blue)
```

The next type of attribute is an ordinal scale—a nominal scale in which a total ordering exists among the categories. Interval and ratio scales are the more quantitative scales and add definitions of dimensions, units, and granularity.

This brief description of attribute type was included to allow the reader to understand the examples in this paper. Attributes have additional types and a number of other properties which are explained in (Iscoe, et al., 1992).

Hierarchical Decomposition

Hierarchies are a natural way to view and organize information and, at some level of abstraction, are a part of most object-oriented and knowledge representation languages. Unfortunately, the simplicity of these concepts can sometimes obscure the semantics that a model is attempting to capture. That one's needs dictate one's

ontological choice is a fundamental premise of knowledge engineering. The ability to systematically define a new set of attributes by partitioning the value sets of old attributes and then using these new attributes to reclassify the domain in accordance with the new requirements is an important aspect of our attribute characterization. By preserving the "ontological map" as a component of the attribute, the domain modeler can shift between the differing paradigms modeled by various classes of objects.

Attribute characterization provides a representation and systematic methodology for the partitioning of attributes that facilitates the way they are organized, subdivided, and built into hierarchies. An attribute restriction is a new attribute whose value set and set of applicable relations are subsets of the original attribute.

Creating a new attribute serves the dual purpose of creating a set of views on the old attribute as well as creating a new attribute. Often, new attributes are defined in terms of old attributes by partitioning the original value set and then equating each new attribute value with an element of the partition. As an example, an accounts receivable (AR) system may use the attribute days_to_payment whose value is the average number of days it takes for the client to pay a bill.

```
(days_to_payment:
:type          ratio_scale
:dimension     time
:unit          days
```

From the standpoint of AR applications, a more useful attribute might be:

```
(type_of_payer:
:type          Ordinal_scale
:Ordered_by    lateness_of_payment
:values  (pays_on_time  slow_pay  dead_beat))
```

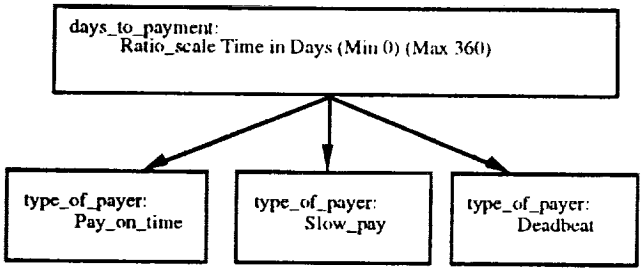


Figure 3 — Partitioning days_to_payment

This new attribute will be defined by partitioning the value set of days_to_payment by subdividing the range of values, then equating each value with one of the elements of the partition as illustrated in figure 3 and described as follows:

```
(type_of_payer
  :mapped_from  days_to_payment
    (pays_on_time   (<=30)
    (slow_pay
        (AND (> 30) (< 90)))
    (dead_beat      (>= 90))))
```

Note that the days_to_payment attribute is based on a quantitative attribute while the type_of_payer attribute is based on a qualitative attribute. In general, an attribute mapping represents a loss of information (in this example, the number of days overdue) in return for a more useful and inherently less detailed category.

Using Population Parameters

Population parameters are used to help automate the process of creating new attributes from old ones. For example, some graduate admissions committees use GRE scores to separate applicants into acceptance categories. Population parameters allow application designers to create new attributes based on restrictions to the original attribute as shown below:

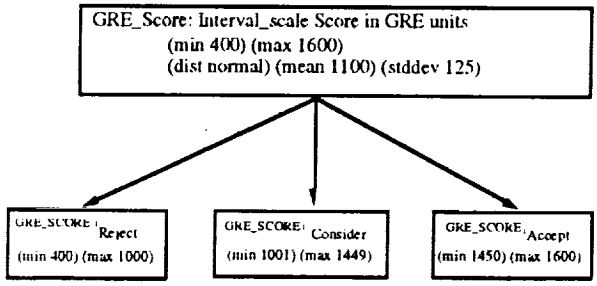


Figure 4 — Using Population Parameters to Restrict an Attribute

Figure 4 shows the GRE score as an attribute which could be attached to a student. Understanding the distribution of values within the value set of GRE scores allows application designers to create partitions in any one of a variety of ways. For example, assume that an application designer wanted to create an initial partition based on the requirement *"accept all students who score in the top x% on the GRE, consider those who score between x% and y%, and reject those who score in the bottom y%."* Given this type of requirement, the domain model contains the appropriate information to use and an algorithm to produce the correct raw score numbers to achieve such a partition.

Another way that these requirements are sometimes stated is to build a partition based on an absolute raw score. For example, a requirement like "*accept all students who score above 1450 on the GRE*" is easily displayed and modeled. Furthermore, this type of specification can be used interactively so that the designer can juggle between raw scores and percentiles until the partitions appropriate for the application domain are produced.

Domain and Specification Models

In this section we focus on relations between attributes within a single domain model class. For the purposes of this discussion we define the following attributes:

```
(Name      :type   identifier)
(Gender    :type   nominal_scale
    :values (male  female))
```

```
(Eye_color  :type   nominal_scale
    :values  (brown, blue, green))
(Benefits   :type   nominal_scale
    :values (Soc_sec, RR, none)
(Age            :type   ratio_scale
    :dimension  (time)
    :unit       (year)
    :granularity  (1)
    :derived    (diff_date cur_date birth_date)
(Medicare_payment :type  ratio_scale
   :dimension  (money)
   :unit       (dollar)
   :granularity  (.01))
   :popparms   ((min 1)(max 10000)(mean 225)))
(Age_m  type:      ordinal_scale
   :values         (under65  65_and_over)
   :mapped_from    age
                   (under65      (< 65))
                   (65_and_over  (>= 65)))
```

Although many other constraints exist, domain model classes can be regarded as consisting of sets of attributes which are either required or might be included within a particular domain model. These constraints are expressed as follows:

must_have(c, a,) — attribute *a* must be used in class *c* in a model.

applicable(c, a) — attribute *a* can be used in class *c* in a model depending on the choice of specification designer.

cond_must_have(c, a, cond) — attribute *a* must be used in class *c* in a model if condition *cond* evaluates to true.

cond_applicable(c, a, cond) — attribute *a* can be used in class *c* a model if condition *cond* evaluates to true.

Within any particular specification model, an attribute is simply classified as used within a class.

used(m, c, a) — within model m, attribute *a* is used in class *c* in model m.

The most straightforward relationship between a domain model and a specification model is that must_have attributes are used in all specification models and applicable attributes are selected by the specification designer. The following rules formalize the semantics of the four constraints on the use of attributes within classes listed above.

(1) must_have(c, a) $\rightarrow \forall$m used(m, c, a)

(2) applicable(c, a) $\rightarrow \exists$m used(m, c, a)

(3) (cond_applicable c a $((p_1\ a_1\ v_1)\ldots(p_n\ a_n\ v_n))$)
$\rightarrow \forall$m, object
[(used m c a) $\rightarrow$
(used m c a_1) $\wedge \ldots \wedge$(used m c a_n) $\wedge$
[(instance m c object) $\wedge$ (in (a object) $\mathcal{U}$(a))
$\rightarrow$ (p_1 (a_1 object) v_1) $\wedge \ldots \wedge$
(p_n (a_n object) v_n)]]

(4) (cond_must_have c a $((p_1\ a_1\ v_1)\ldots(p_n\ a_n\ v_n)$)
$\rightarrow \forall$m,object
[(used m c a_1)$\wedge\ldots\wedge$(used m c a_n)
$\rightarrow$ (used m c a) $\wedge$
[(p_1 (a_1 object) v_1) $\wedge$
$\ldots \wedge$
(p_n (a_n object) v_n) $\wedge$ (instance m c object)
$\rightarrow$ (in (a object) $\mathcal{U}$(a))]]

For example, in a domain model, name might be required for all specification models, while eye_color could be selected only if it were appropriate for the particular specification model.

```
(person
   :must_have ((Name ())
   :applicable ((eye_color ())
   ...)
```

The application of these constraints when *cond* is vacuously true is a fairly standard feature in most modeling languages of this type. However, name and eye_color are attributes which are total functions and are not as interesting as the cases that occur when the attributes are partial functions.

Conditions for Function Evaluation

Recalling that an attribute is a function which maps objects to a particular property, *cond* can be interpreted as the condition which must be satisfied for the attribute to be a total instead of a partial function. In other words, *cond* defines the subset which is the domain of applicability of the partial function. For example for a person class medicare_payment is only applicable if age is 65 or over and benefits is none.

```
(cond_applicable  person  Medicare_payment
    ((= Age_m 65_and_over) (= Benefits none)))
```

The domain modeling system is designed so that the conditions required to establish the proper domain for an attribute are automatically maintained. These conditions are constrained in such a way that tractability is maintained and are of the form $((p_1\ a_1\ v_1)\ldots(p_n\ a_n\ v_n))$, where p_i is the name of a predicate, a_i is the name of an attribute, and v_i is a value of the attribute.

A user can create a specification model with any particular class hierarchy as long as the domain policies and constraints are satisfied.

We are currently experimenting with ways to capture and verify domain modeling constraints by presenting redundant information in a variety of ways. We believe that many of the specification problems in large systems are created when value set changes cause a single case to be changed but fail to correct cases that were identified from a previous inference.

For example, if we assume that Medicare_payment is only applicable if age is 65 or over and benefits is none, the system can infer that Medicare_payment cannot apply to a person who is younger than 65.

In fact, assume
(cond_applicable person Medicare_payment
((= Age_m 65_and_over) (= Benefits none))),
then
∀m, object
((used m person Medicare_payment) →
(used m person Age_m)∧(used m person Benefits)∧
((instance m person object) ∧
(in (Medicare_payment object) [1 10000])
→ (= (Age_m object) 65_and_over) ∧
(= (Benefits object) none))) (5)

After Medicare_payment is used in a model, if user is trying to assign a Medicare_payment to a person who is younger than 65, using rule (5) will lead to a contradiction.

A key point is that when people are presented with value sets they automatically and unconsciously perform substitutions such as the ones listed above. This is a reasonable way to build a model until a value set changes. In large systems, value sets are frequently changed. Consequently, conclusions that were drawn by using negation to infer values become invalid. We use the applicability of conditions and the system's knowledge of value sets to attempt to provide the proper cases for the domain modeler to check when conditions change.

Discussion

In this paper, we have presented the concept of modeling application domains in order to achieve the operational goals of program specification, code generation, and reverse engineering. The main concept is that multiple specification models can be created that are consistent and "correct" with respect to a domain model. Domain models represent information about a particular industry area. Specification models represent information about a particular system.

The middle oval on the right side of figure 1 represents the process of code generation through program transformation. Given a specification model, executable code can be generated by performing a series of correctness-preserving transformations on the specification. The goal of this part of the project, which is not yet active, is to produce efficient code that satisfies the original specification.

Domain and specification models are constructed by using a graphical interface to interactively create a set of rules based on attribute value set partitions and the preceding axioms. The system is being implemented using Motif GUI on SPARC workstations. Although it is currently operating in a single user mode, it is being designed to be accessed simultaneously by multiple domain modelers. We are also trying to accelerate the knowledge capture process by reverse engineering data models that have been captured by an existing EDS case tool and instantiating them into the appropriate domain models.

Acknowledgments

We wish to thank Betty Milstead and Raman Rajagopalan for their comments on earlier drafts of this paper.

References

Amarel, S. 1968. "On Representations of Problems of Reasoning About Actions," in *Machine Intelligence 3*, D. Michie, Ed., American Elsevier, New York, pp. 131-171.

Barstow, D. 1985. "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1321-1336.

Barstow, D. 1988. "Artificial Intelligence and Software Engineering," in Shrobe, H., Ed., *Exploring Artificial Intelligence*. AAAI. Morgan Kaufmann, San Mateo, CA.

Bledsoe, W. W., and Hines, L. M. 1980. "Variable Elimination and Chaining in a Resolution-Base Prover for Inequalities," *Proceedings of the 5th Conference on Automated Deduction*, Les Arcs, France, Springer-Verlag, pp. 70-87.

Borgida, A., Brachman, R.J., McGuinness, D.L., and Resnick, L.A. 1989. "CLASSIC: A structural data model for objects," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pp. 59-67.

Curtis, B., Krasner, H. and Iscoe, N. 1988. "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268-1287.

Davis, R. 1991. "Knowledge Representation: Broadening the Perspective," AAAI-91 Panel, Anaheim, CA.

Iscoe, N., Browne, J.C., Werth, J., and Liu, Z.Y. 1992. "Attributes – Building Blocks for Modeling Application Domains," Submitted to IEEE TSE.

Iscoe, N., Williams, G. and Arango, G., Eds. 1991. *Domain Modeling for Software Engineering, Proceedings of Domain-Modeling Workshop*, Austin, Texas.

Lenat, D.B. , Guha, R.V., Pittman, K., Pratt, D., and Shepherd, M. 1990. "Cyc: Toward Programs with Common Sense," *CACM*, vol. 33, no. 8, pp. 30-49.

Liu, Z.Y. and Farley, A. 1991. "Tasks, Models, Perspectives, Dimensions," *The 5th International Workshop on Qualitative Reasoning* Austin, Texas, pp. 1-12.

Van Sickle, L. 1992. "Reconstructing Data Integrity Constraints from Source Code," *Proceedings of Workshop on Artificial Intelligence and Automated Program Understanding*, Tenth National Conference on Artificial Intelligence, San Jose, CA.

Description of Research Interests and Current Work Related to AUTOMATING SOFTWARE DESIGN

Hermann Kaindl
SIEMENS AG Österreich, PSE
Geusaugasse 17, A-1030 Vienna, Austria

Research Abstract

While I am working in industry in a department dedicated to *software engineering*, major part of my research dealt with various aspects of *artificial intelligence*. As can be seen from the enclosed list of selected and recent publications, my research interests include heuristic search, machine learning, knowledge acquisition and knowledge-based systems. Moreover, I performed applied research in the areas software engineering and human-computer interaction.

Recently, I became more and more interested in combining methods from these areas, for instance we used *hypertext* for improving the process of knowledge acquisition. Moreover, I emphasize the relationship between the fields, for instance the relations between AI frames and objects in object-oriented approaches. I think that there are many issues in common in knowledge acquisition and object-oriented analysis. Generally, the task of building knowledge-based systems appears to me to include many aspects of software engineering.

Partly, we develop conventional as well as knowledge-based software for telecommunications, and partly we work for the European Space Agency. While we did not really get to the point of building domain-specific software design systems yet, I completely agree that domain-specific knowledge plays a major role in developing software. For instance, the functionality of the software for one satellite is typically not so much different from that of the software for the next satellite. I feel that improvements in the general software development process (e.g., object-oriented approaches) will have to be combined with the use of large domain-specific knowledge bases.

Selected Bibliography

- Köll, A., and Kaindl, H., "A New Approach to Dynamic Weighting", to appear in *Proc. Tenth European Conference on Artificial Intelligence (ECAI-92)*, Vienna, August 1992, Chichester, England: Wiley.
- Kaindl, H., and Scheucher, A., "Reasons for the Effects of Bounded Look-Ahead Search", to appear in *IEEE Transactions on Systems, Man, and Cybernetics (SMC)*, 1992.
- Snaprud, M., and Kaindl, H., "Knowledge Acquisition Using Hypertext", to appear in *Expert Systems with Applications*. Earlier versions are available in *Proc. World Congress on Expert Systems*, Orlando, Florida, December 1991, 781-788, New York: Pergamon Press, in *Proc. AAAI-91 Workshop on Knowledge Acquisition*, Anaheim, CA., July 1991, and in *Proc. Artificial Intelligence and Knowledge-Based Systems for Space*, ESTEC, Noordwijk, May 1991.
- Kaindl, H., and Ziegeler, H.G., "Object-oriented Approaches, Frames, and Access-Oriented Programming", to appear in *Object-Oriented Programming in AI* (Scott Woyak und Zhongmin Li, Eds.), AAAI Press.
- Lercher, L., and Kaindl, H., "Problems, Communication, and Common Sense", to appear in *ACM SIGART Bulletin*.
- Kaindl, H., and Ziegeler, H.G., "Reasoning Types and AI Programming Paradigms", to appear in *Software Engineering and Knowledge Engineering (IJSEKE)*. An earlier version is available in *Proc. Third International Conference on Software Engineering and Knowledge Engineering (SEKE'91)*, June 1991, 96-101.
- Kaindl, H., and Ziegeler, H.G., "HIS—An Information System about Hypertext on Hypertext", to appear in *ACM SIGLINK Newsletter* 1.
- Kaindl, H., Shams, R., and Horacek, H., "Minimax Search Algorithms with and without Aspiration Windows", *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-13(12), 1991, 1225-1235.
- Kaindl, H., and Snaprud, M., "Hypertext and Structured Object Representation: A Unifying View", in *Proc. Third ACM Conference on Hypertext (Hypertext '91)*, San Antonio, Texas, December 1991, 345-358. An earlier version is available in *Proc. Fourth International GI Congress on Knowledge-Based Systems*, Munich, Germany, October 1991, 231-242, Berlin: Springer-Verlag.
- Kaindl, H. (Ed.) "*Proc. Seventh Austrian Conference on Artificial Intelligence*", Vienna, Austria, September 1991. Berlin: Springer-Verlag.
- Mehlsam, G., Kaindl, H., and Barth, W., "Feature Construction during Tree Learning", in *Proc. Fifteenth German Workshop on Artificial Intelligence (GWAI-91)*, Bonn, Germany, September 1991, 50-61, Berlin: Springer-Verlag.
- Shams, R., Kaindl, H., and Horacek, H., "Using Aspiration Windows for Minimax Algorithms", in *Proc. Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney,

Australia, August 1991, 192-197, Los Altos, Calif.: Kaufmann.
- Kaindl, H., and Ziegeler, H.G., "Comparing object-oriented programming, frames, and access-oriented programming", in *Proc. AAAI-91 Workshop on Object-Oriented Programming in AI*, Anaheim, CA, July 1991.
- Kaindl, H., and Ziegeler, H.G., "Knowledge-Based Systems: Their User Interface and Dependability", in *Proc. IFIP Workshop on Dependability of Artificial Intelligence Systems (DAISY_91)*, Vienna, Austria, May 1991, 53-62, Amsterdam: North-Holland.
- Kaindl, H., and Ziegeler, H.G., "HyperAuthor—An Authoring Tool Based on Hypertext", in *Proc. Hypertext/Hypermedia '91*, Graz, Austria, May 1991, 156-163, Berlin: Springer-Verlag.
- Ziegeler, H.G., and Kaindl, H., "A Cyclic Pattern Resulting from a Constraint Satisfaction Search", in *Proc. CAIA-91: Seventh IEEE Conference on Artificial Intelligence Applications*, Miami Beach, Florida, February 1991, 337-344. An earlier version has been presented at the *AAAI-90 Workshop on Constraint Directed Reasoning*, Boston, Mass., July 1990.
- "Tree Searching Algorithms", in *"Computers, Chess, and Cognition"* (T. A. Marsland and J. Schaeffer, Eds.), 133-158, New York: Springer-Verlag, 1990.
- Kaindl, H., and Ziegeler, H.G., "Knowledge Acquisition for a Configuration Task", in *Proc. AAAI-90 Workshop on Knowledge Acquisition*, Boston, Mass., July 1990.
- Kaindl, H., and Ziegeler, H.G., "Some Aspects of Knowledge-Based Configuration", in *Proc. AVIGNON '90 - Expert Systems & their Applications--Artificial Intelligence, Telecommunications & Computer Systems*, Avignon, May/June 1990, 41-54.
- Scheucher, A., and Kaindl, H., "The Reason for the Benefits of Minimax Search", in *Proc. Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, August 1989. 322-327, Los Altos, Calif.: Kaufmann.
- "Portability of Software", *SIGPLAN Notices* 23(6), 1988, 59-68.
- "Minimaxing: Theory and Practice", *AI Magazine* 9(3), 1988, 69-76.

Appendix: Technical Biography

Hermann Kaindl received the Dipl.-Ing. degree in computer science in 1979 and the Doctoral degree in technical science in 1981, both from the Technical University of Vienna in Austria.

Since 1984, he has been a lecturer on artificial intelligence at the Technical University of Vienna, and in 1989, he received the *venia docendi* for "Praktische Informatik", which is comparable to tenure. He is currently with the department of Program and System Engineering, Siemens AG Österreich, where he leads software projects and is in charge of a group of software engineers. His research interests include planning and search, machine learning, knowledge acquisition, knowledge-based systems, as well as certain aspects of software engineering and human-computer interaction.

Dr. Kaindl is a member of the Austrian Society for Artificial Intelligence, the American Association for Artificial Intelligence, and the International Computer Chess Association.

Automating the Design of Scientific Computing Software

Elaine Kant
Schlumberger Laboratory for Computer Science
P.O. Box 200015
Austin, Texas 78720–0015 USA
kant@slcs.slb.com

Abstract

SINAPSE is a domain-specific software design system that generates code from specifications of equations and algorithm methods. This paper describes the system's design techniques (planning in a space of knowledge-based refinement and optimization rules), user interaction style (user has option to control decision making), and representation of knowledge (rules and objects). It also summarizes how the system knowledge has evolved over time and suggests some issues in building software design systems to facilitate reuse.

Introduction

SINAPSE is a domain-specific software design system that generates code from specifications of equations and algorithm methods. Our goal is for SINAPSE to be a practical program-synthesis system that solves a restricted class of problems. In particular, we aim to reduce mathematical modelers' programming chores by enabling modelers to specify programs at the level at which they are described in technical papers.

A trend toward three-dimensional modeling (previously too expensive to attempt for many applications) is both making programs more complex and requiring implementation on parallel architectures (for acceptable performance). Both consequences of this trend argue strongly for automatic code generation – to avoid errors in programs and to save modelers from having to learn about rapidly changing architectures. Because efficiency of code and interfacing with other codes are factors for many of our users, the code generation system must be understandable and modifiable.

The current SINAPSE implementation focuses on one class of algorithms – finite difference methods for solving partial differential equations. We have used the system to generate about a dozen families of programs for solving acoustic wave propagation problems of interest to Schlumberger modelers. With these programs (for which no comparable hand-coded versions existed), the modelers have achieved new results in the application areas. However, all the programs were specified by knowledgeable users, and we manually optimized critical code sections after experimenting with the automatically generated program. Current research involves generating more efficient code and making the system more easily accessible to modelers.

Although we primarily apply the system to finite difference problems, we have also generated several rather different types of codes and have used subsets of the system in other applications. Approximately half of the system (consisting of the synthesis framework and an array-level language to target code translation) is independent of the domain, although focused on scientific computing. We have used this part of the system to generate some geometric modeling codes, starting from an array-level specification language.

The lessons from SINAPSE are similar to those of other knowledge-intensive systems: it is important to design representations that are close to the users' mental models; abstraction of concepts is important; and rules and objects provide useful representation techniques. An emerging concern is how to encourage more sharing among software design systems. The last section of this paper suggests that reuse of components and reasoning algorithms may be possible among different software design systems themselves.

Specializing Design Techniques

The driving force in the implementation of SINAPSE has been the collection of design techniques appropriate for our applications. The classes of design techniques as well as the problem itself then determine the types of user interaction that are required. Finally, the knowledge representation is strongly suggested by the reasoning techniques and user interface requirements.

Given our fairly narrow application domain and goal of practical program synthesis, the most appropriate design technique is knowledge-based refinement, including the application of optimizing transformations. Refinement choices are made by heuristics or modeler specification. Although our approach includes knowledge-based optimization, as the performance demands on synthesized code have scaled up, we have seen more need for traditional types of optimization such as code motion supported by data-flow analysis.

We have explicitly chosen not to use some types of reasoning techniques. For example, learning about choices in context and learning about run-time code performance might eventually be appropriate, but we chose not to address learning, discovery, or complex

search issues in the current system. We also do not attempt inference by theorem proving; this would require very detailed domain models before any progress could be made, and these formalisms would make it difficult to allow the kinds of not-strictly-correctness-preserving approximations that modelers frequently make. However, we are attempting to develop a clean characterization of the semantics of the synthesis constructs. This is a good guideline for domain analysis and helps make the meaning of the constructs independent of the implementation. A clean semantics makes a construct easier to explain to users and easier for developers to modify.

The states in the problem space in which SINAPSE operates include descriptions of (partially implemented) programs and facts about the specifications and implementations. The space is navigated by carrying out sequences of synthesis tasks. Originally we tried to streamline the problem-solving mechanism by letting the actions in program synthesis carry out the navigation, with design choices being presented to the user as needed, but this proved confusing to the users and difficult to modify. Therefore we are moving towards an explicit plan representation. We expect to conclude by declaratively encoding the set of goals about program function and performance, plans for achieving those goals, and control knowledge about which plans to use for different circumstances. The plans consist of partially-ordered (sub)goals, bottoming out at actions that include asking the user for information, applying program refinement rules, and applying program optimizations.

A specification in SINAPSE is a collection of design decisions, most of which can be thought of as control information about which program refinements to make, or which facts to declare. In addition, sometimes a specification actually defines a new refinement and then asserts that the new alternative is the refinement that should be made.

SINAPSE is implemented in *Mathematica*[Wol88]. *Mathematica* is both an algebraic manipulation system, useful for scientific programming, and a programming language with modern features such as a pattern language and rules. Other implementation languages would also be reasonable choices, but *Mathematica* allows us to have everything in one language in which our target users are comfortable.

Phases of Design

In order to make the system comprehensible to developers and end users, and to encourage collaboration with others, we have divided the software design process into a series of phases:

- **problem set up**
- **algorithm synthesis**
- **program optimization**
- **target code generation**

How common these stages are in other design systems for scientific computation is an open question, but evidence for them can be found in [PC91; AEH+89; Coo90]. A more detailed, though somewhat dated, description of these phases is given in [KDMW91].

The first phase, **problem set up**, involves helping the user define the problem. The result should be a set of equations such as would be described in a modeling article. In our application, problem set up is accomplished by working through a network of choices (goals and tasks) that set up the equations to be solved. For applications about which SINAPSE is knowledgeable, it presents parameterized equation generators; otherwise the user must define the equations mathematically.[1] Mathematical formalization, when the equations are not given directly by the user, involves a relatively straightforward knowledge-based expansion. Next, SINAPSE may reformulate the equations via simplification, normalization, and redundant equation elimination. Other reformulations, such as averaging of material values, depend on user specification. *Mathematica*'s algebraic manipulation is especially useful at this stage.

The problem set up phase should probably be viewed as three distinct phases. Two, which are independent, are describing the **physical model** in general terms, reusable for a number of specific problems, and describing the **target properties** of the computing environment in which a specific problem must be solved. Properties of the target environment might include machine architecture (such as type of parallelism available) and limitations on run time and storage space. A specific **problem description** would then be the next phase, that would customize a physical model to a specific set of knowns and unknowns (and any desired interpretation or analysis of the computed results) and might modify the equations to be used based on the specification of target environment properties.

SINAPSE's **algorithm synthesis** begins with selecting an algorithm schema corresponding to the modeler's design decision(s) and then filling in the details. This level includes all the domain-specific computing knowledge that an applications expert would have, typically the numerical approximation methods to be applied to the equations. The types of implementation decisions are those that would be reported in a detailed technical article. At the end of this phase, programs will be expressed in *Psiam*, an array-level language that we are developing. The search for effective combinations of design decisions is currently left to the user if the default choices are not acceptable. Program details are filled in by refinement rules. Elaboration of the design decisions often involves the use of algebraic manipulation for computing approximations. If desired, the modeler can specify fragments of code directly in *Psiam*. The schema instantiation may involve elaborating parts of code such as initializations or outputs that eventually need to migrate to other sections

[1] In other applications, such as mechanics and circuits problems, systems often have more detailed descriptions of the physics of the systems and tools to instantiate the physical laws in a specific problem. The instantiation often involves much unguided object slot filling rather than the guided, dependent, goal satisfaction used in SINAPSE.

of code. The migration is done too explicitly now; we will evolve to a more general mechanism with partial orderings and data flow analysis.

Performance choices are made at the next stage, **program optimization**. This level includes all the types of knowledge that any good scientific programmer should know regardless of the application domain. Some examples of design decisions made at this stage are store vs. recompute decisions, data structure selection (array representation, primarily space compression techniques), and the corresponding operator implementations. Data and control parallelism from the domain have been explicitly represented and maintained through the program transformations until, at this level, parallelism is either exploited or, for target languages not supporting it, expanded into looping constructs or sequentialized. A number of optimizing transformations are applied. To support the data structure selection and optimizations, there is some inferencing to determine data types of dimensions, properties of arrays, and simplifications of conditionals (for example, to transform conditionals on array indices into loops with specific bounds). Currently SINAPSE uses special case reasoning for such inferences; it would benefit from an interface with an inequality prover and probably other provers or decision procedures.

The result of expansions of the previous step is expressed in *MathCode*, another language that we have developed. *MathCode* is a procedural language that abstracts away from Fortran and C constructs but has almost no remaining implementation freedoms. The final phase of **target code generation** from *MathCode* is accomplished by a recursive-descent parser with action rules for each different target language.

Interacting with Users

Our initial concern in user interaction was simply to ensure that modelers could specify their problems and override SINAPSE's default design decisions. A SINAPSE specification, which contains a set of design decisions, might "ideally" contain just decisions at the level of specifying the problem. In reality, of course, the system does not have enough information to make all the algorithm and implementation choices. Even when the system thinks it has enough knowledge, not all modelers will agree with the choices. The evolution of these aspects of the interface will be discussed here. Some other issues concerning the modeler's interface to scientific codes are outside the scope of this discussion. For example, while our total environment will involve an interface for specifying the geometry of the world being modeled and an interface for visualization of the results, these are largely separate research efforts.

The philosophy of partitioning the problem-solving load between the user and SINAPSE was discussed in [Kan90]. The conclusions, to which we still subscribe, can be summarized by:

- SINAPSE should structure the problem-solving sessions because people are smarter than software design systems and can adapt; however, SINAPSE should present the user with significant decision points and alternative implementation choices that match problem-solving models.
- SINAPSE should cooperate by making suggestions (heuristics about appropriate choices, help in finding similar specifications or concepts); however, people should have ultimate veto power over system choices.
- SINAPSE should be able to explain, at least minimally, specification choices and decisions that have been made.
- SINAPSE should have a system for helping users and developers add new knowledge.
- SINAPSE should share knowledge bases so progress for any purpose (synthesis, explanation, knowledge acquisition, or system integration) is tested by and contributes to progress for all purposes.

Current Interface

Currently, the user must be reasonably knowledgeable to set up a SINAPSE specification. Specifications are usually made in a text file that is loaded at the beginning of a session, but most choices can also be specified interactively with simple menus (for enumerable choices) or fill-in-the-blank interfaces. Also, although program fragments can be specified at the array level (effectively defining new refinement rules at specification time), there is no interactive support for this. In the interactive mode, the user can request text string explanations of the decision issues, alternatives, and system heuristics. Answers provided by the user are checked against legitimate patterns. In addition, the user can confirm or modify interactively the choices suggested by system heuristics or a previously loaded text-file specification. SINAPSE can write out a text file of the decisions made interactively, or made by a mix of previously specified text and interactions.

We have begun to make SINAPSE more accessible to modelers. We are adding pointers to examples of specific choices and their realization in target programs based on our demo suite. A graphical interface with modern menus, multiple status and help windows, and hypertext navigating is being implemented, and a minimalist-style user manual is being written. Because of the large number of design decisions and the different classes of anticipated users (some modelers care more about approximation method choices, some about efficiency of implementation), we also will need a mechanism to control which design decisions are visible to the user. One possibility is to make visibility dependent on the phase in which the decisions are made and on whether the decisions are based on hard constraints (forced choices) or heuristics or simple defaults.

Declarative Decision Structures

A good interface is critically dependent on the correctness and understandability of the underlying domain models. Indeed, users cannot even write text-file specifications if they do not have a good understanding of what needs to be specified. Although we have had some difficulty in explaining how the system works to different domain experts, the specification language

seems to be converging as we gain better understanding of the domain. Earlier versions of SINAPSE did not have all decisions explicitly represented, but we are adding a definition mechanism that ensures that all design decisions are properly inserted in a global task network. Correctly representing the domain means not only having the right set of design decisions, but ordering the decisions sensibly and representing dependencies between decisions. Although SINAPSE was able to generate the same set of programs with a more unstructured representation, having a good, declarative representation of the decision structure turns out to be critical for acquiring a specification, for storing out a specification in text format for later use, and for explaining specifications and system decisions to users.

Dependencies between Decisions

An explicit representation of all dependencies between design decisions would be useful for helping the user understand what must go into a specification, for recording specifications made interactively, and for replaying revised specifications. For example, the dependency network helps the user understand that a particular decision may not even be relevant unless some other set of choices has been made. SINAPSE distinguishes between user-specified decisions and decisions inferred by the system based on those decisions. Only decisions in the first class need be recorded in the text-file specification. Decisions in the second class can be made again automatically if the specification is resubmitted. This argument assumes a static synthesis system. If more alternatives for a decision are added at a later date, the existing heuristics may no longer force a choice. Hence, it might also be useful to record the full history of inferences to help the user augment the specification in the face of an evolving system.

Currently, synthesis times are all under 20 minutes, and the decision making portions are usually on the order of minutes, so simply recording the primary decisions and recomputing the rest has been acceptable and it has not seemed necessary to build a full-fledged truth maintenance system. We do have a simple dependency network that records definitions and uses of synthesis facts. Because we wish to record decision dependencies for purposes of explanation, at some point the expense of building an incremental change system may be justified.

Because the user can help make implementation decisions, we also foresee a need for representing dependencies between user specifications. This general phenomenon of specifications accommodating to implementations is discussed in [Swa82]. One example that we have seen in SINAPSE is that a modeler may combine periodic and taper boundary declarations to implement an absorbing boundary condition when the target language is SIMD Connection Machine Fortran (to enable the use of an efficient circular shift operation). Even if a boundary isn't really periodic, the tapering operation makes the effective boundary value nearly zero on both edges, which means declaring the boundary to be periodic is not harmful. These dependencies should be recorded because if the target architecture is changed, we want to reconsider the choices of periodic and taper boundaries (even though both were user-specified) in the light of the new architecture.

Ordering Decisions

Users are sensitive to the order in which specification decisions are made; this order must make sense to them. Ordering is constrained by dependencies between decisions. In general, of course, the ordering of the decisions will follow the ordering of the phases described in the previous section, with implementation decisions such as data structure representations following problem set up specifications such as boundary conditions. However, some details can vary with the application. For example, in some cases all dependent variables may depend on the same independent variables so it might make sense to define independent variables first and then list dependent variables. In other cases, it might make more sense to define each dependent variable in terms of its specific independent variables. To support this, SINAPSE can present a different set of design decisions with alternative orderings for different applications.

Currently, when used in the interactive mode, the SINAPSE system presents the design choices in a linear sequence, and modelers do not always understand why a particular ordering is used. It would help considerably if we represented the *partial* ordering on the design choices, with a user interface that allows specification according to the partial ordering rather than an arbitrary linearization of that ordering. We do believe however that the system should explicitly present the decisions in the partial ordering rather than expecting the user to write the decision in arbitrary order in a text file or to navigate around a large collection of objects and to know what properties must be filled in or what commands must be issued. We plan to experiment with a graphical depiction of the decision network that is actively modified as choices are made.

Explanation

Representing information about decisions could help generate good explanations for how to set up specifications or why the system made the specific choices [WMK92; Swa83]. It both cases, a likely priority is: most heavily weight the choices involving problem description decisions (user choices before system choices), then the state of the implementation design so far, then the user's generic preferences, then the system's heuristic rules, and finally the system defaults.

Representing the Knowledge

The representation of knowledge in Sinapse has been discussed elsewhere [KDMW91] and so will not be repeated in detail here. We simply note that our goals for code generation and user interaction suggest that our knowledge representations be declarative, object-oriented descriptions of design choices and algorithm schemas. The object-oriented representation for design constructs includes the use of multiple inheritance,

with a small number of fairly flat hierarchies for algorithm type, application type, and so on. As discussed earlier, since the initial system design, the importance of more explicit goals and plans for the user interface has become clear. In addition to the declarative representations, there are procedural languages that can be used in describing programs: *Psiam* at the array level, and *MathCode* at the imperative level. The semantics of *Psiam* are still evolving; *MathCode* is the most mature and stable of all the representations. *Mathematica*'s pattern matching and symbolic simplifications are useful in defining transformation rules for both refinement (elaboration) and optimization. Recently we have also added a mechanism to record some of the major transformation steps (by transformation name and by before and after states). While we do not expect to record every single transformation step, we expect to eventually have more control over transformation applications; currently most are just anonymous *Mathematica* rules that fire whenever they match rather than being explicitly applied. Most likely there will be named sets of transformation rules that are applied at specific phases.

Evolving the Knowledge

To measure the evolution of knowledge in SINAPSE, for the past 16 months we have kept records about changes to the system. A regression test suite is maintained so that changes can be tested for compatibility and compared for performance. Although the records are only as good as the effort people put into keeping them and more careful analysis is need, some rough generalizations can be made.

Overall, the total system has grown steadily. The initial effort, before detailed records were kept, was mostly in adding domain knowledge and very primitive code generation knowledge. Since that time, we have focused on generating efficient code for multiple target languages and architectures, on adding domain knowledge that fills gaps in our application domain, and on making the system more understandable via additional explicit knowledge about design decisions and explicit representation of dependencies between decisions. There have been no huge waves of expansion and compression of the entire system representation, although individual components do grow and shrink as knowledge is added or more concisely represented.

Some basic information about size may give a general picture of the evolution of knowledge. The current system is now more than 20,000 lines of *Mathematica* code, a 38% increase over the system of 16 months ago. The declarative representations of the domain knowledge and problem-solving structure have grown the most – from 13% to 19% of the system, a 111% increase. There are currently about 100 types of design facts of the fill-in-the-blank form and 33 menu-choice decisions with an average of 3 alternatives. There are currently about 60 program-synthesis tasks; as well as adding new tasks, the ordering among the tasks has been refined over time. Procedural knowledge about how to refine domain descriptions to algorithms and coding constructs has grown only 15% and slipped from 41% to 35% of the system. (No count on the number of rules or functions is available. This is a place where the content of the knowledge has increased, but the representation has gotten more concise, so the overall growth looks low.) Knowledge about code generation has increased 35%, but as a percentage of the entire system held almost even, moving from 30% to 29%. (Much of the work that has gone into code optimization is not complete and is not reflected in the version of the system described here. The code-optimization techniques will add approximately 5,000 more lines of code.) The program-synthesis framework, while growing 54%, has only gone from 16% to 17% of the total system. The growth has been in the areas of mechanisms for the expanded knowledge about synthesis tasks and the recording of major transformation steps.

Of the 360 recorded changes to the system (in terms of number of entries, not number of lines of code or numbers of facts involved), 30% have involved changes to the internal representation or knowledge about the program-synthesis process, 15% have been changes visible in the human interface, 15% have been changes to domain knowledge, 35% changes to programming knowledge (reflected in the generated code), and 5% to the operating system interface. Overall, 20% of these changes were described as new capabilities, 24% as generalizations of existing capabilities, 20% as bug fixes, 5% as efficiency improvements, 28% as improvements in the clarity of the system or the code it generates, and 5% as other.

The frequent occurrence of changes to improve representation clarity reflects both improved understanding of the domain and deficiencies in the original representations of design goals and actions. Improvement is still needed in expressing dependencies between decisions, both the order required by the decisions, in terms of definition-use chains, and task-ordering preferences. We also expect it would be useful to be able to express a difference between hard constraints (forced choices), heuristics based on available information, and default choices (based on no information).

Analyzing the types of changes that are made should help us determine what sort of knowledge acquisition tools we should build. At present only a minimal number of rudimentary knowledge-building aids exist in SINAPSE. They help inspect the structure of synthesis tasks and dependencies and check for completeness of information about design decisions. Based on analysis of the changes and conversations with modelers, we have identified a small number of knowledge-acquisition activities that we would like to support more automatically for end users as well as for developers. These activities include the addition of new approximation operators, of variations on input/output handling, of new algorithm schemas, and the packaging of existing algorithms inside user-defined outer loops.

Sharing among Design Systems

The amount of knowledge required for automating software design is very large, even for quite restricted

classes of problems. The automated software design community would be likely to make faster progress if it explored the possibilities of reuse *among* design systems as well as reuse within a single domain-specific system. How do we design our systems to facilitate this sharing? Possibilities include reuse of system components (some domain-independent), reuse of reasoning algorithms, and reuse of interface languages (such as a *Psiam*-like array-level language). Similar proposals have been made before of course, such as generic tasks for expert system building blocks [Cha86], compositional modeling for engineering modeling [FF91], standardization work in the knowledge-representation community, and the suggestion of working out theories for program synthesis [Smi91].

Reuse of system components might be possible if we could divide systems into components with well-defined interfaces. This means we first need to agree on the *meaning* or content of any specification languages or intermediate representations. We also need to formalize the *form* of the interfaces. Ironically, the methodology for figuring out how to implement a specified need in terms of existing components, or how to adapt components to a function, will probably itself exploit automated software design techniques. Some components may be large, some may be clusters of knowledge about well-defined concepts.

In SINAPSE we are attempting to identify some major phases in the design of scientific computing software and to provide different languages for some of the levels. The languages may vary to exploit mathematical formulations, array-manipulation, and conventional applicative languages so that specifications can be entered in the most convenient style. Next, we need to determine whether these stages make sense for other applications. Within these levels, there might be formalizations of abstractions such as coordinate transforms, pointers, I/O, and parallelization. Ideally, SINAPSE would then be able to interface to other systems, for example to generate a different target language, or call subroutines rather than generate code for specific tasks.

The reasoning-technique (shell) approach is another cut at providing tools. We might ask what sorts of tools for different reasoning strategies would be useful for automating software design. For example, SINAPSE could use someone else's inequality prover, or an outside tool for analyzing data flow, or an expression optimizer to minimize operator costs according to a declarative cost model or to order for optimal numerical stability. It would be useful to have language-independent compiler optimization tools.

If we could find a useful set of common tools or components, major barriers (besides the not-invented-here syndrome) might be standardizing the interfaces and achieving portability of tools. Even though it is now possible to interface many different languages, in a system with multiple implementation languages, the overhead in both execution and modifiability can be quite high. Neverless, even if it requires reimplementation, a clearly specified set of tools and algorithms for accomplishing the goals of the tools should facilitate reuse.

Acknowledgements

Current and past members of the SINAPSE project who should be recognized for their work on the concepts and implementation of the system include Ira Baxter, Hung-Wen Chang, François Daube, Bill MacGregor, and Joe Wald. Many thanks to Ira Baxter and Ursula Wolz for comments on drafts of this paper.

References

H. Abelson, M. Eisenberg, M. Halfant, J. Katzenelson, E. Sacks, G. J. Sussman, J. Wisdom, and K. Yip. Intelligence in Scientific Computing. *Communications of the ACM*, 32(5):546–562, May 1989.

B. Chandrasekaran. Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design. *IEEE Expert*, 1(3):23–30, Fall 1986.

G. O. Cook. ALPAL, a Program to Generate Physics Simulation Codes from Natural Descriptions. *International Journal of Modern Physics*, 1(1):1–55, 1990.

B. Falkenhainer and K. D. Forbus. Compositional modeling: finding the right model for the job. *Artificial Intelligence*, 51:95–143, 1991.

E. Kant. Human and Computer Responsibilities in Program Synthesis. In *Workshop Notes–Knowledge-Based Human-Computer Communication*, pages 65–67, Stanford, CA, March 1990.

E. Kant, F. Daube, W. MacGregor, and J. Wald. Scientific Programming by Automated Synthesis. In M. R. Lowry and R. D. McCartney, editors, *Automating Software Design*, chapter 8, pages 169–205. AAAI Press/The MIT Press, Menlo Park, CA, 1991.

R. S. Palmer and J. F. Cremer. SIMLAB: Automatically Creating Physical Systems Simulators. Technical Report TR 91-1246, Department of Computer Science, Cornell University, Ithaca, New York, November 1991.

D. Smith. Theory-Based Support for Software Development. In *Workshop Notes–Automating Software Design: Interactive Design*, pages 162–165, Los Angeles, CA, July 1991.

W. Swartout. On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, 25(7):438–440, July 1982.

W. Swartout. XPLAIN: A System for Creating and Explaining Expert Consulting Systems. *Artificial Intelligence*, 21(3):285–325, September 1983.

U. Wolz, K.R. McKeown, and G.E Kaiser. Automated Tutoring in Interactive Environments: A Task-Centered Approach. In M.J. Farr and J. Psotka, editors, *Intelligent Instruction by Computer, theory and practice*. Taylor and Francis, Washington DC, 1992.

S. Wolfram. *Mathematica: a System for doing Mathematics by Computer*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

Domain Specific Software Design for Decision Aiding

Kirby Keller and Kevin Stanley
McDonnell Aircraft Company
McDonnell Douglas Corporation

McDonnell Aircraft Company (MCAIR) is involved in many large multi-discipline design and development efforts in the production of tactical aircraft. These involve a number of design disciplines that must be coordinated to produce a integrated design and successful product. Our interpretation of a domain specific software design (DSSD) is that of a representation or framework that is specialized to support a limited problem domain. Figure 1 contrasts domain specific vs. domain independent approaches. A DSSD is an abstract software design that is shaped by the problem characteristics. This parallels the theme of objected-oriented analysis and design[1] of letting the problem model directly drive the design. The DSSD concept extends the notion of software reusability to include representations or frameworks. It supports the entire software life cycle and specifically leads to improved prototyping capability, supports system integration, and promotes reuse of software designs and supporting frameworks.

Initial prototyping is improved if one can start development with a framework that is suited to the characteristics of the problem. This framework can be specialized as the development evolves to provide a more efficient means for the domain expert to prototype The effect is to shorten the distance between the domain expert and the working prototype by providing a domain language to state requirements and supporting automated code generation. This concept of a supporting framework can be extended to the systems level. Multi-discipline design efforts may require the integration of individual DSSDs which are critical to concurrent engineering efforts. Domain specific designs that capture problem solving representations can be leveraged in future work. These designs offer flexibility by addressing a problem domain and hence are a better starting point for reuse than particular application modules. It may also be possible to create libraries of such designs that can be matched to problem characteristics.

The example presented in this paper is the task network architecture or design which was developed for the MCAIR Pilot's Associate program. The task network concept supported both module development and system integration within the domain of operator decision aiding. It is presented as an instance where a software design exhibited many of the attributes associated with DSSD concept. The Pilot's Associate program (contract #F33615-86-C-3802) was sponsored by the Defense Advanced Research Projects Agency and administered by the United States Air Force. More recent work in this area has been performed in conjunction with McDonnell Douglas Research Laboratories and Michigan State University.

Pilot Decision Aiding Example:
As part of the Pilot's Associate (PA) program, McDonnell Aircraft (MCAIR) Company developed and demonstrated an "associate" system for tactical aircraft performing an air-to-ground battlefield interdiction mission. The demonstrated mission functionality included threat assessment, system capabilities assessment, threat reaction planning,

[1] Rumbaugh, et al, Object-Oriented Modeling and Design, Prentice Hall, 1991.

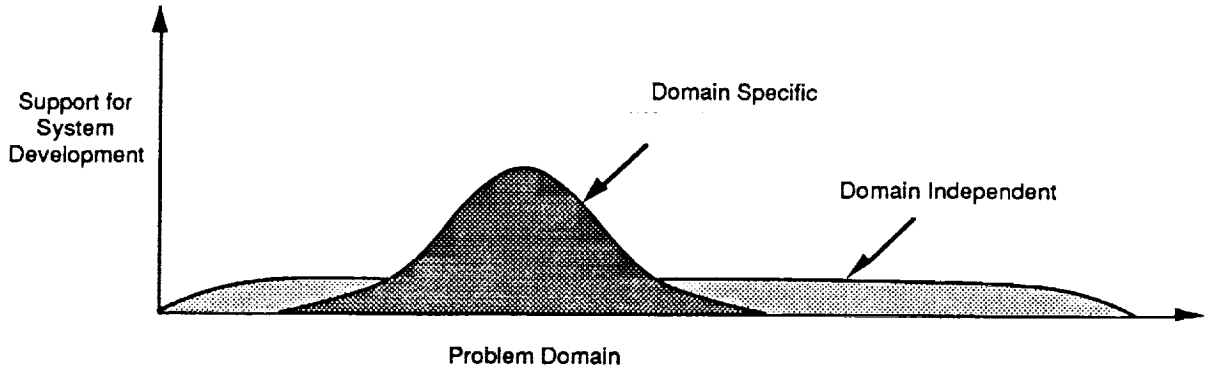


Figure 1. Domain Specific Software Design Provides Improved Support for Specific Problem Domains.

target attack planning, pilot monitoring, and information management. Appropriate controls and displays were developed to support the demonstration in a manned aircraft dome simulator. The system development approach and software architecture is based upon a task network system model. The activities of the pilot, PA and external agents such as a wingman are modelled by objects called tasks. Tasks may be decomposed into a complex sequence, or network, of more detailed subcomponents. This model of the task sequences and their functionality define a hierarchical network of tasks which allow the representation of complex system and pilot activity for both steady state behavior and reaction to changes in the environment. It captures dependencies and interactions between activities and provides a means for overall control of the PA problem solving process. The structure derived from the task network system model provides: 1) a domain specific requirements language or representation that is shared by the domain expert and software developer, 2) data structures and frameworks for the software design, and 3) visibility into the system behavior that helps create a more intuitive interface and system operation.

The top-level architecture of the task network framework is shown in Figure 2. The main components of the framework are: input packet post-processing, the context model, the task network mission model, exception handling and task execution. Data flow in this architecture consists of communication from external processes through packet post processing which appropriately manipulates the data to update objects in the context model. Events are signalled to the task network mission model, resulting in changes in task status or the execution of an exception handler. When tasks are activated they are placed on a task agenda and executed in order of priority. The execution of tasks may result in the modification of internal models (internal actions) or the communication of data to other processes (external actions).

The task network is modelled after the procedural network structure, first proposed in the NOAH system[2]. The partially ordered sequence of tasks in the network identifies control flow and context information for the state of the mission. Through an explicit representation of system and pilot tasks, the system may reason about it's own

[2] Sacerdoti, Earl D., A Structure for Plans and Behavior (Elsevier: Computer Science Library, 1977).

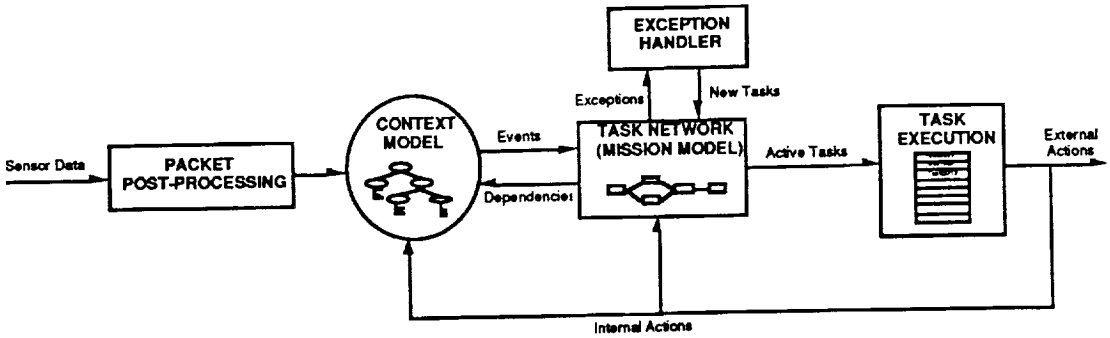


Figure 2. Task Network Top-Level Architecture

activities. Typically, this reasoning involves predicting system timeliness, interactions between tasks, errors of omission by an external agent (e.g. a pilot), the information requirements of the pilot, or responses to failure conditions of the task. Each task is represented as a specialist responsible for performing a function when activated.

Tasks are defined in a hierarchical manner such that they may be decomposed into subtasks which refine the activities that they represent. This is useful for reasoning about tasks at different levels of abstraction in monitoring, planning, and execution. The task network provides mechanisms for:

1) system coordination,
2) maintaining assumptions about the environment,
3) handling exceptions, and
4) representation of interaction with the pilot.

The task network allows dependencies to be placed on states of the environment, the pilot, and the PA system through a subset of the task network framework referred to as the Context Model. The Context Model is developed as an hierarchical, distributed, object-oriented database. It is used to represent information about the external environment, the pilot, the aircraft, and the PA system itself. This representation was designed to allow the detection of events from state data. Since PA has relatively little control over actions in the external environment (e.g. hostile threats, weather, etc.), it must make many assumptions during plan generation and execution. PA must be able to adapt quickly and correctly when the external environment changes in a way that invalidates the planned system behavior. Dependencies allow the tasks to represent complex relationships between the tasks and the state of the environment (i.e. the context of the current situation). These dependencies are checked when changes are made to the Context Model parameters. When dependencies are violated, this is signalled to the task. This signal is referred to as an exception.

Exceptions represent violated dependencies which require a response by the system. This response is referred to as an exception handler. These exception handlers are defined for tasks to aid in the recovery of violations in the assumptions of the plan. Exception handlers may be either local or system exception handlers. Local handlers are implemented using methods on the task which result in minor, local changes to the plan or states of various systems. System handlers involve the creation of System Response Plans which use the task network framework as a control mechanism for replanning portions of the currently executing task network.

The task network architecture is a domain specific design in that it is a framework

that provides support for requirements specification, design, and development at the module and system level. The benefits of the task network architecture are realized from a set of features which aid in the development of an application which lies in the real-time decision support domain. The components of the architecture support system integration by providing a uniform representation of the elements of the domain. These components and their inter-relationships were developed to address the requirements of the PA domain but it has been implemented as a explicit framework that is readily applicable, in part or in whole, to problems with similar characteristics. These features are described in the following sections.

Explicit Representation of System Plans

The requirements of the PA system are often described in terms of the aircraft mission. This mission description includes the objectives of the pilot and his weapon system in a hostile and uncertain environment. Mission decomposition is usually performed using a number of representative scenarios. This mission decomposition is a key characteristic of the domain. Mission decomposition is a top-down approach for dissecting a combat mission into its functional segments. These functional segments are then divided into the tasks which are required to complete each segment. As functions and tasks become more specific, they can be analyzed in terms of information flow and functional partitioning. The task network supports this specification through it's explicit representation of the sequence of tasks in the mission.

The explicit representation of functions as tasks in the system provides advantages in software design by supporting graceful adaptation through reasoning about task timeliness, the explicit representation of parallelism in task execution, by promoting modular coding techniques, explicit control synchronization between tasks, and visibility into system operation through the use of mnemonic names for tasks.

Enables Control Reasoning

Completing tasks by their assigned deadlines is the very definition of a hard real-time system. However, the character of the Pilot's Associate prompted us to expand the definition to include the concepts of both hard and soft deadlines. While meeting hard deadlines is a requirement for correctness, meeting soft deadlines is not strictly required, but is certainly desirable. Control reasoning is useful for a decision support system which is attempting to optimize its performance outside of hard scheduling constraints. The system may predict missed deadlines, delete unnecessary steps to meet imminent deadlines, and perform reasoning about solution quality/timeliness trade-offs. Control reasoning is also supported by the management of system priorities on tasks.

Supports Coordination and Cooperation

Knowledge partitioning is a natural and inevitable approach to the design and development of large systems. The PA system was partitioned into modules, each of which is a knowledge based system with the possibility of concurrent execution. While concurrency may not be utilized physically, the components of the PA are intended to operate in a functionally distributed fashion. Functional distribution, in this context, merely means that the components are designed to allow the possibility of concurrent operation. Each component is a real-time system. That is, each component receives events and data asynchronously and carries out steps of assessment, planning and execution, all constrained by timing requirements. For such a collection of real-time knowledge based systems to form an integrated system, they need to behave in a coordinated manner that is also timely, responsive, and adaptive to a changing environment. Coordination refers to a system-wide coherence among tasks and

plans, to a resource management scheme based on a global perspective, and to dynamic adjustment of tasks and plans to accommodate changes in overall system performance goals.

Opportunistic Execution of Tasks
Quite often in system design, the correct sequence of execution of system tasks is unknown. The task network representation allows the specification of incomplete temporal constraints on control flow. The non-linear plan representation allows ambiguity of task ordering. The execution of parallel tasks may be performed opportunistically and behavior is situationally dependent. This allows the system to improve and tune it's performance based on the context of the current situation.

Exception Handlers Modify Behavior Through Changes in Explicit Plans
The PA problem domain is dynamic and hostile. Subsequently, plans may be expected to be invalidated quite often. This adds complexity to the system requirements and design. The Task Network Architecture handles this through an explicit link between environmental data and tasks referred to as dependencies. Exception handlers are procedures which are implemented to respond appropriately to events. Each task is responsible for handling these events by one of several classes of reactions such as: abandoning the task, retrying to achieve the results of a task, choosing an alternate method for accomplishing the task results, or repairing the cause of the error. The complexity of exception handlers may be quite simple, or may require extensive replanning of the mission.

Replanning and Execution Are Interleaved
It is not possible to predict the time that events impacting the mission will be encountered. Deliberation on new plans often involves extensive processing resources devoted to solving problems encountered in the execution of plans. However, a real-time system cannot afford to halt execution while replanning is underway. Due to this, the system must be capable of replanning portions of the mission, while completing unaffected portions. The current design of the task network allows the system to inhibit the execution of tasks which are in an exception state, while continuing to execute other tasks which are unaffected by the exception.

Control Flow Manipulated Graphically
One of the tools available to software developers for managing complexity is that of graphical interfaces. The partially ordered sequence of tasks lends itself very well to a graphical depiction of the sequence of tasks performed during the mission. The implementation of tools for the graphical manipulation of tasks provides an efficient and intuitive interface for system control specification. At the same time, these tools will also provide aid in debugging the performance and functionality of the system since the current state of the system is represented pictorially through the state of the tasks in the network.

One of the key features of the task network approach is the ability to describe tasks from the perspective of a mission, and then use that same description as a foundation for code development. This philosophy is supported by the encapsulation of functionality as provided by object-oriented programming. The most efficient means of designing and modifying a task network data structure is through the use of a graphical interface which allowed for direct manipulation[3] of the task network. The task network

3 Hutchins, E.L, Hollan, J.D., and Norman, D.A (1986). Direct Manipulation Interfaces in D.A. Norman, W.S. Draper (Eds.): User Centered System Design: New Perspectives in Human-Machine Interaction, Hillsdale London: Lawrence Erlbaum, 1986).

implementation offers a mechanism by which application code could be seamlessly integrated with code generated via these graphic descriptions.

Requirements Specification Language

The task network framework is a programming paradigm for the development of intelligent systems. The task network architecture provides support through the entire software development process, from requirements generation (specification) through maintenance as shown in Figure 3. Each module function is developed using the task network framework for planning, assessment, and human interface functions. The goal of the framework is to provide a common language between the requirements specifier, system designer, and system user. This will lead to systems which have traceable requirements in the program design and whose operation may be easily understood by the user. The program structure serves as a model of the user in performance of the mission. The network of tasks describe the sequence of tasks to be performed by the system and user. Unplanned events must also be accounted for in the system design. The design for the detection of unplanned events, the dependency mechanism, makes the conditions for plan failure explicit.

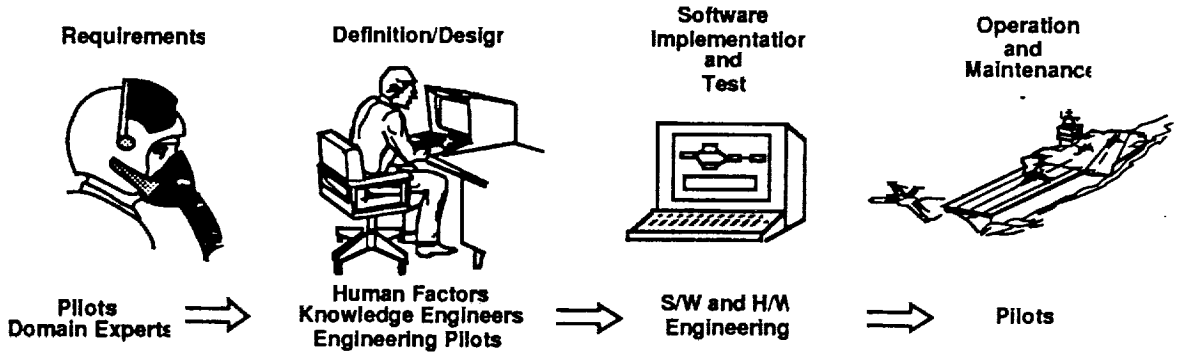


Figure 3 - Software Model Supports the System Life Cycle

The analysis of the mission results in identification of pilot and system activities as they relate to various phases of the mission. This analysis includes the identification of mission objectives, tasks that need to be performed, information required to perform the mission successfully, candidate approaches for automation and decision aiding support, and the identification of constraints imposed by combinations of the above.

Sequencing tasks in the mission identifies the context within which tasks are to be performed and the temporal constraints for efficient and effective mission performance. Through the process of mission decomposition, functional requirements are identified along with the context in which they are to be executed. This is a result of the representation of the mission sequence--mixing pilot and system activities together in a coordinated fashion.

User Activity Model

Interactive decision support systems must provide more aid than they require in user attention to the system. The primary goal for modeling the user in the task network architecture is to minimize interactions between the system and the user and thereby develop a non-intrusive, cooperative decision support framework. Through modeling the user, the system is supplied with necessary context

sensitivity to work efficiently with the user.

The pilot monitoring approach which was adopted, focussed on the state of the world represented in the Context Model, rather than on explicit pilot interaction. The approach isolates the monitor from the need to identify all methods of performing a task, all actions that may undo a given task, and explicit legal time intervals for tasks. The result is concise, robust, task-monitoring rules that can be incrementally enhanced as the Context Model grows richer. The task network represents the activities of the user by activating tasks when evidence indicates that they are being performed or have been completed. Active tasks identify activities which are being performed by the user which may be used to identify the information which is required by the user of the system. This provides a mechanism for providing both timely, and relevant information to the user.

Issues:

The major benefit of DSSD promises to be the creation of a library of reusable designs which can be classified by problem characteristics or domain to which they are applicable. An application developer could then quickly piece together a development framework from these designs. What is needed is an enumeration of the fundamental designs and a description of the range of domains that they cover.

The DSSD concept supports the notion that the initial prototyping effort should be directed at establishing or assembling a design for the particular application. This will allow leveraging the representations/frameworks associated with component DSSDs. The development of a application design based on existing DSSDs should be a goal in order to achieve system modularity, reuse, and development efficiency (eg. automated code generation).

Traditionally the press for real-time performance tends to drive designs toward system representations that are flat and efficient at the expense of rich representations which support the management of design complexity and effective interface design. It becomes a matter of development costs vs. the need for a real-time design.

The integration of DSSDs to support and integrate different design disciplines is a key to the application of the DSSD idea to large systems. In the PA example, the task network is used as a means to analyze the human factors elements of information management and automation, threat assessment, mission and tactical replanning, and as a means to determine the effect of system failures on mission activities. A focus on the concepts of DSSD should result in frameworks for integrating lower level module designs into a more coherent system design.

Knowledge-Intensive Software Design Systems: *Can too much knowledge be a burden?*

Richard M. Keller

Sterling Software
NASA Ames Research Center - Artificial Intelligence Research Branch
Mail Stop 269-2, Moffett Field, CA 94035-1000

(415) 604-3388 (Phone); 604-3594 (FAX); Keller@ptolemy.arc.nasa.gov

Abstract

While acknowledging the considerable benefits of domain-specific, knowledge-intensive approaches to automated software engineering, it is prudent to carefully examine the costs of such approaches, as well. In adding domain knowledge to a system, a developer makes a commitment to understanding, representing, maintaining, and communicating that knowledge. This substantial overhead is not generally associated with domain-independent approaches. In this paper, I examine the downside of incorporating additional knowledge, and illustrate with examples based on our experience building the SIGMA system. I also offer some guidelines for developers building domain-specific systems.

1. Introduction

One of the long-prevailing tenets of artificial intelligence research is that "knowledge is power" -- the more knowledge made available to a system, the better. The knowledge-based software engineering (KBSE) community, as evidenced by its self-designation, embraces this philosophy no less than other disciplines within AI. Traditionally, the knowledge represented and used by practitioners of KBSE has been knowledge about the programming discipline, itself. Increasingly, however, researchers are recognizing the utility of representing and using knowledge about the target programming domain (e.g., business, manufacturing, science, telecommunications, engineering, etc.) to facilitate automation of various facets of the software engineering process [1,2,3]. In fact, the seductive "knowledge is power" maxim has even found a receptive audience in the mainstream software engineering community, where several workshops on the topic of "Domain Modeling" have been held over the past few years [4].

The migration toward domain-specific systems comes as no great surprise. Despite progress in developing general-purpose methods for automated software engineering [5], the practical application of these techniques has met with limited success. In some cases, these methods have failed to scale up appropriately; in other cases, the methods have proven too mathematically-sophisticated to appeal widely to the practicing community of software engineers. However, by incorporating additional domain knowledge and constraints, it becomes possible to specialize and simplify these methods to a point where they are more tractable and less daunting to apply. While acknowledging the considerable benefits of domain-specific, knowledge-intensive approaches to automated software engineering, it is prudent to carefully examine the costs of such approaches, as well. In adding domain knowledge to a system, a developer makes a commitment to understanding, representing, maintaining, and communicating that knowledge. This substantial overhead is not generally associated with domain-independent approaches. In this paper, I examine the downside of incorporating additional knowledge, and question whether adding knowledge introduces as many new problems as it solves.

Over the past several years, I have been involved in the development of a domain-specific software design system for scientific modeling. To ground my remarks, I will briefly describe this system and its knowledge requirements. Then I will describe some of the additional burden placed on the developers as a result of the knowledge-intensive nature of this system. Finally, I will attempt to generalize from our experience and present some guidelines and caveats for others developing domain-specific KBSE systems.

2. SIGMA : A knowledge-based scientific software environment

The goal of the SIGMA project [6] is to provide computational support for scientists engaged in computer modeling and simulation of physical systems. Examples of such systems include planetary atmospheres, forest ecosystems, and biochemical systems. Generally, these systems can be modeled as a set of algebraic and ordinary differential equations, where the terms in the equations interrelate the physical quantities of interest. Although computer models play a crucial role in the conduct of science today, scientists lack adequate software engineering tools to facilitate the construction, maintenance, and reuse of modeling software. Usually, scientific models are implemented using a general-purpose computer programming language, such as FORTRAN. Because this type of general-purpose language is not specifically customized for scientific modeling problems, the scientist is forced to translate scientific constructs into general-purpose programming constructs. This manual translation process can be very complicated, labor-intensive, and error-prone. Furthermore, the translation process obfuscates the original scientific intent behind the model, and buries important assumptions in the program code that should remain explicit. The resulting software is typically complex, idiosyncratic, and difficult for anyone but the primary scientific author to understand.

We are building a knowledge-based software environment that makes it easier for scientists to construct, modify, and share scientific models. The SIGMA (Scientists' Intelligent Graphical Modeling Assistant) system functions as an intelligent assistant to the scientist. Rather than construct models using a conventional programming language, scientists will be able to use SIGMA's graphical interface to "program" visually using a more natural high-level graphical data flow modeling language. The terms in this modeling language denote scientific concepts (e.g., physical quantities, scientific equations, and datasets) rather than general programming concepts (e.g., arrays, loops, counters). The scientist-user interacts with the system to construct a syntactically and semantically valid data flow graph, such as the one illustrated in Figure 1. In this graph, the lettered nodes represent scientific quantities, such as temperature, pressure, and density. These quantities are input to scientific equations (depicted by numbered nodes in Figure 1) which calculate output quantities.

The data flow graph in Figure 1 represents part of a planetary atmospheric model developed at NASA Ames Research Center [7]. The model computes the temperature (T) at some altitude point above a planetary surface based on input data (r) measuring the extent to which a radio signal refracts upon penetrating the atmospheric gases at that altitude.

Although visually simple, the graph masks a number of non-trivial technical problems that must be addressed to actually execute the corresponding program. For example, the input refractivity value is a vector quantity, not a scalar, so there is an implicit iteration being performed. Note also that Equation # 4 is a differential equation that must be numerically integrated to solve for P. In addition, the scientific units specified for the various inputs to an equation may not be compatible and must be

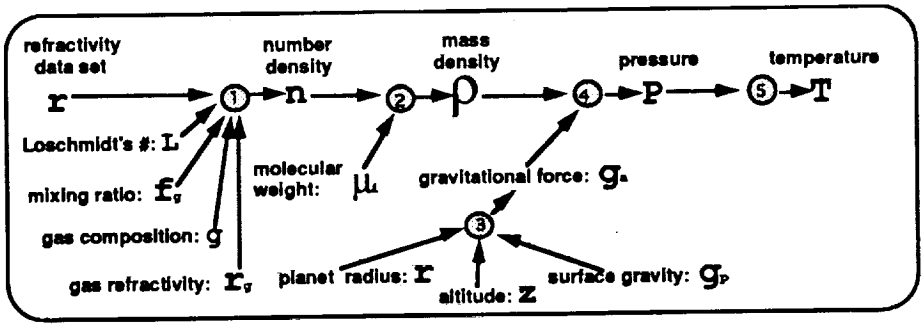


Figure 1: Data flow graph representing a portion of a planetary atmospheric model. Letters represent physical quantities. Numbered circles correspond to equation application nodes.

converted to a common unit system before that equation can be applied. SIGMA's interpreter handles these details automatically for the user.

On the surface, SIGMA appears similar to a large class of data flow based visual programming environments that have been developed recently. These systems help users graphically construct software in a variety of application areas, including image processing and scientific visualization (Khoros/Cantata [8], Iconicode/IDF [9], AVS [10], apE [11]), scientific instrument design (LabVIEW [12]), and simulation (STELLA/IThink [13], Extend [14]). In all of these cases, however, the software tool has fairly limited knowledge of the application domain. Although the tools enforce simple syntactic checks on the data flow graphs and perform some type-checking, none of these tools has a deep semantic understanding of what the data flow program is doing and whether the operations on the data make sense. As a result, it is possible with these tools to create a syntactically valid flow graph that is semantically meaningless to a domain specialist. In contrast, SIGMA assists the scientist during the model-building process and checks the model for consistency and coherency as it is being constructed. In particular, SIGMA's domain knowledge assists the system in interpreting the user's intentions and in constructing a semantically meaningful program.

SIGMA is closer in spirit to ϕ_0 [15]. ϕ_0 is a domain-specific automatic programming system constructed to assist in generating oil well log interpretation software. The system was designed for direct use by petroleum scientists, who would use it to construct geological models expressed as a set of quantitative equations relating geological parameters of interest. Like SIGMA, ϕ_0 makes extensive use of scientific domain knowledge to aid in the program synthesis process. The next section describes SIGMA's domain knowledge.

3. SIGMA's Domain Knowledge

SIGMA's domain knowledge is represented and stored in a hierarchically-structured, frame-based knowledge base of over 500 concepts which contain information about scientific equations, physical quantities, scientific units, numerical programming methods, scientific domain concepts, and bibliographic citations. A partial overview of the knowledge base is depicted in Figure 2.

SIGMA's knowledge can be partitioned into four categories:

1. **Cross-disciplinary scientific knowledge:** General knowledge available to persons with a scientific background, including knowledge about various physical quantities, scientific domain objects, scientific measure units, foundational equations, and scientific handbook data.

2. **Area-specific scientific knowledge:** Quantities, domain objects, equations, and data pertaining to a specific scientific discipline (e.g., biology, ecology, physics).

3. **Problem-specific knowledge:** Domain objects and relations pertaining to the specific physical system being modeled by the scientist.

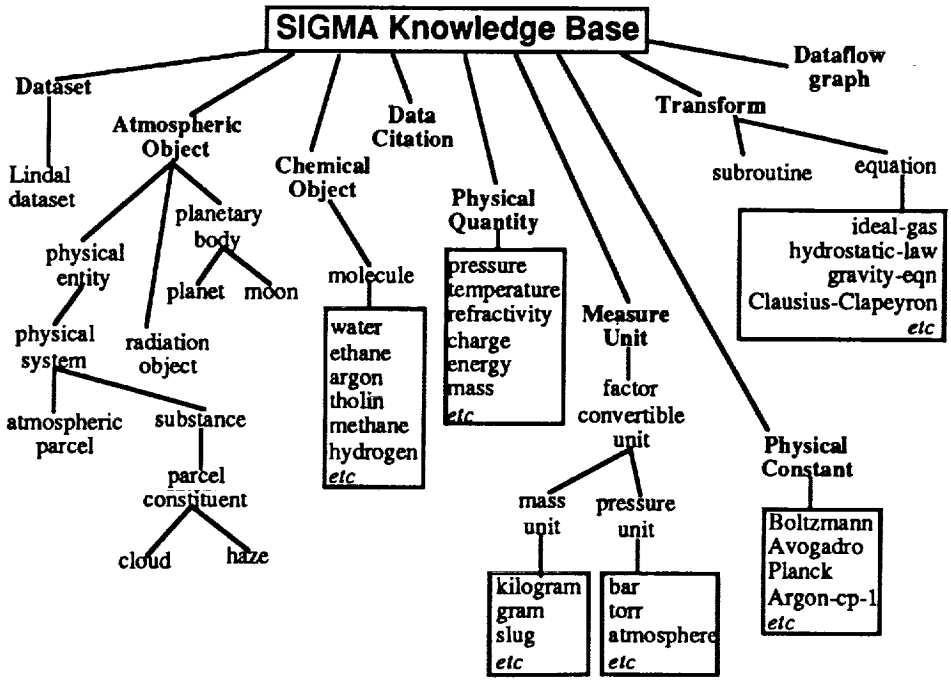


Figure 2: Overview of SIGMA's knowledge base

4. **Programming knowledge:** Knowledge about numerical programming methods, data structures, control, etc. (In the current version of SIGMA, much of this knowledge is implicit in the data flow interpreter.)

Although a detailed discussion of SIGMA's knowledge base and representational structures is outside the scope of this paper, I will briefly describe one of the key elements: SIGMA's equation representation.

Each SIGMA equation consists of a syntactic equation formula plus a semantic interpretation for each of the symbols in the formula. Each symbol is identified with an attribute of some class of domain objects in SIGMA's knowledge base. The domain objects associated with the various equation symbols are constrained to obey specified relationships among each other. Consider Figure 3, which illustrates how Equation 1 of Figure 1 is represented internally within SIGMA. Equation 1 states that the number density (n) of a gas mixture (i.e., the number of particles per volume of mixture) is equal to the refractivity index (r) of the entire mixture divided by a weighted sum of the refractivity indices (r_g) of the individual gases within the mixture.

As shown in Figure 3, the semantics of this equation are represented in terms of the domain objects that the equation interelates, namely the gas mixture (called an atmospheric-parcel), the homogeneous pure-gas subcomponents of the mixture (called constituents), and the individual gases that are included in the mixture. The symbols "r" and "n" in the equation are linked to the refractivity and number-density attributes of the same atmospheric-parcel. The subscript "g" is identified with the constituents attribute of that same atmospheric-parcel. The constituents attribute stores a pointer to each constituent within the atmospheric-parcel. The symbol "f_g" is linked to the mixing-fraction of a constituent, and stores the percentage of this constituent as a fraction of the total quantity of gas within the atmospheric-parcel. The symbol "r_g" represents the refractivity attribute of a gas that is contained by the constituent. Finally "L" refers to a physical-constant called Loschmidt's Number.

In essence, this representation provides a set of domain constraints that must be satisfied for the equation to apply legitimately in a given domain situation. As a scientist builds up a data flow graph such as the one in Figure 1, he or she is unknowingly constructing an invisible constraint network of domain objects and relations similar to the one illustrated in Figure 3. This constraint network provides a sound semantic interpretation for the graph.

4. SIGMA's Knowledge Burden

The rationale behind our decision to invest considerable time and energy into representing domain knowledge for SIGMA is simple and, we believe, compelling: How can a machine interact intelligently and synergistically with a scientist to create modeling software if the machine has no understanding of the scientific problem under study? Without this shared understanding, SIGMA would have to rely on user guidance for many of the functions it now performs automatically. Our users have expressed an impatience with systems that need to be "spoon-fed"; given an option, they would rather drop down into FORTRAN and code the model themselves! Our only alternative, it seems, is the knowledge-intensive route.

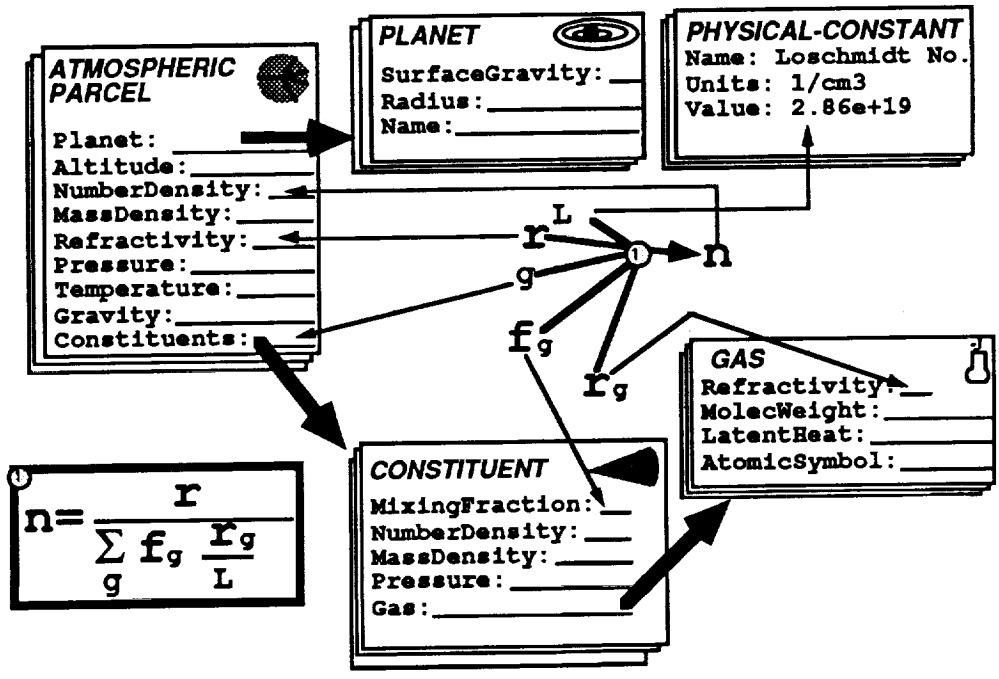


Figure 3: Representation for Equation 1 in Figure 1.

The Catch-22 in this situation is that the addition of domain knowledge imposes burdens on the developer, maintainer, and users of the interactive software design system:

- **The Comprehension Burden:** System developers must analyze and understand the application domain and the class of problems to be solved.

  Our experience with SIGMA is that a significant amount of time (several person-months of effort) is required to sufficiently understand the scientific modeling problems presented by our collaborators in planetary and ecosystem sciences. Of course the difficulty is a function of many variables, including the developer's prior background knowledge and experience in the application domain, the caliber of expert advice and guidance, the complexity of the scientific modeling problem, etc.

- **The Representation Burden:** Developers must design suitable representations to capture the knowledge.

  In our experience, the problem of representing domain knowledge is a significant modeling problem in itself. Within SIGMA, we have identified a need for representing quantities, quantitative and qualitative relationships, part-whole and subsumption relationships, temporal and spatial relationships, modeling assumptions, and other difficult representational constructs. A comprehensive treatment of all of these issues is beyond the scope of any single project. (However, see [16] for an ambitious effort in this vein.)

- **The Maintenance Burden:** System maintainers or users must add new knowledge, update old knowledge as it becomes outdated, and generally maintain the integrity of the knowledge base.

  For example, novice and intermediate SIGMA users will want to enter new equations and new physical quantities into the system. Sophisticated SIGMA users may wish to modify the original domain theory that was captured and encoded as a by-product of discussions with our expert collaborators. In fact, the domain theory (i.e., the domain objects, attributes, and relations) is as much a part of the scientist's model as the equations. Because the equations are intimately linked to the underlying domain theory (as discussed in Section 3), entering a new equation is complicated, and modifying the domain theory has wide-ranging implications. As a result, the current version of SIGMA does not permit users to modify the domain theory.

- **The Communication Burden:** Developers must implement tools and techniques that adequately convey the system's knowledge to the user, and vice versa.

  Consider once again SIGMA's equation representation. It is non-trivial to convey this type of a representation scheme to a naive user without exposure to knowledge-based or object-oriented techniques. Building an adequate user-friendly editor for SIGMA will be a challenging (and no doubt time consuming) task. Navigating and editing the concepts in the knowledge base pose similar difficulties.

Although these problems are significant, most of them are pose no greater or lesser challenge than those faced by developers, maintainers, and users of *any* sophisticated knowledge-based system. Software engineering, after all, is just another application area for knowledge-based techniques.

5. Easing the Burden

Despite the extra effort involved, and the new problems introduced, I still believe it is worth the effort to incorporate domain knowledge as an integral part of an automated software engineering environment. I believe the newly-introduced problems are challenging, but tractable. And without incorporating additional knowledge, I see no way to provide more intelligent and domain-sensitive tools to practitioning software engineers. In this spirit of pragmatism, I offer the following recommendations to those building knowledge-intensive, domain-specific tools:

- **Generality:** Keep the knowledge base and the representations general, without going overboard. This will facilitate entry of new information into the knowledge base, and encourage reuse of existing knowledge and representational constructs in new, similar domains.

- **Stability:** Choose an application for which the domain knowledge is relatively stable. This will minimize the maintenance burden.

- **Scope:** Choose an application for which knowledge is well-circumscribed, yet broad enough to make the endeavor worth your effort. If the knowledge can be reused in other applications, the development costs can be amortized over a shorter period of time.

- **Content:** Choose an application for which the domain theory is well-understood and commonly accepted. This will simplify the process of building an acceptable domain theory and reduce maintenance and communication costs.

- **Terminology:** Use vocabulary that is as familiar as possible to users. This will ease the communication burden.

- **Grainsize:** Avoid modeling phenomena in more detail than necessary for the task -- unless warranted due to generality and subsequent reusability.

Of course the developers of software systems do not always have control over the selection of an application domain. In this case, the above recommendations can be used to evaluate the suitability of domain-specific approaches with respect to a particular domain.

6. Conclusion

Yes, I still believe in the "knowledge is power" axiom. But more than ever, I feel it is important to heed its most-overlooked corollary: "There is no such thing as a free lunch". Caveat emptor!

Acknowledgments

Thanks to the SIGMA group, and especially to Michal Rimon, who implemented the current version of our system. Thanks also to Pandu Nayak who provided us with his RML representation language.

References

[1] D.Barstow, "Domain-Specific Automatic Programming", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, pp. 1321-1336, Nov. 1985.

[2] E.Kant, F.Daube, W.MacGregor, and J.Wald, "Scientific Programming by Automated Synthesis", in *Automating Software Design*, pp. 169-206, M.R.Lowry and R.D.McCartney (eds.), AAAI Press, Menlo Park, CA, 1991.

[3] D.Setliff, "On the Automatic Selection of Data Structure and Algorithms",in *Automating Software Design*, pp. 207-226, M.R.Lowry and R.D.McCartney (eds.), AAAI Press, Menlo Park, CA, 1991.

[4] N.Iscoe, "Domain Modeling -- Evolving Research",*Proc. Sixth Annual Knowledge-Based Software Engineering Conference*, pp. 234-236, IEEE Computer Society Press, Los Alamitos, CA, 1991.

[5] M.R.Lowry and R.Duran, "Knowledge-Based Software Engineering", chapter in *Handbook of Artificial Intelligence, Vol. IV*, A.Barr and P.Cohen (eds.), Addison-Wesley, New York, 1989.

[6] R.M.Keller and M.Rimon, "A Knowledge-based Software Development Environment for Scientific Model-building", AI Research Branch technical report #FIA-92-12, NASA Ames Research Center, Moffett Field, CA, forthcoming July 1992.

[7] C.P.McKay, J.B.Pollack, and R.Courtin, "The Thermal Structure of Titan's Atmosphere", *Icarus*, vol. 80, pp. 23-53, 1989.

[8] Khoros/Cantata software product, Khoros Consortium, EECE Department, University of New Mexico, Albuquerque, NM.

[9] Iconicode and IDF software products, Iconicon, Palo Alto, CA.

[10] AVS software product, Stardent Computer, Inc., Sunnyvale, CA.

[11] apE 2.0 software product, Ohio Supercomputer Center, Columbus, OH.

[12] LabVIEW software product, National Instruments, Austin, TX.

[13] STELLA and IThink software products, High Performance Systems, Lyme, NH.

[14] Extend software product, Imagine That, Inc., San Jose, CA.

[15] D. Barstow, R. Duffey, S. Smoliar, and S. Vestal, "An Overview of Φnix", in *Proc. National Conference on Artificial Intelligence* (AAAI-82), pp.367-369, Pittsburgh, PA, August 1982.

[16] R.V.Guha and D.B.Lenat, "Cyc: A Mid-Term Report", *AI Magazine*, 11(3), 1990.

Automating Software Design System DESTA

Vladimir A. Lovitsky
Associate Professor
Software Engineering Department
Institute of Radioelectronics
Kharkov, Ukraine
(0572) 409 113 (Fax)

Patricia D. Pearce
Professor, Head of Computing Department
University of Plymouth
Drake Circus, Plymouth
Devon, PL4 8AA, UK
pat@uk.ac.psw.cd
(0752) 232 541 (Office)
(0752) 232 540 (Fax)

Abstract

"DESTA™ " is the acronym for the *D*ialogue *E*volutionary *S*ynthesizer of *T*urnkey *A*lgorithms by means of a natural language (Russian or English) functional specification of algorithms or software being developed.

DESTA represents the computer-aided and/or automatic artificial intelligence "forgiving" system which provides users with software tools support for algorithm and /or structured program development.

The *DESTA* system is intended to provide support for the higher levels and earlier stages of engineering design of software in contrast to conventional CAD systems which provide low level tools for use at a stage when the major planning and structuring decisions have already been taken.

DESTA is a knowledge-intensive system. The main features of the knowledge are *procedures, functions, modules, operating system commands, batch files, their natural language specifications* and *their interlinks.*

The specific domain for the *DESTA* system is a high level programming languages likes Turbo Pascal 6.0.

The *DESTA* system is operational and runs on a IBM PC computer.

1. Introduction

At present software development is the biggest obstacle to major new breakthroughs in computing. The biggest limitation in software development is the failure of imagination that people tend to project: *"A user only really knows what he wants when he sees a finished attempt"*.

How we develop software at present. We tend to develop software in the same way we did it in the 1960s i.e. it's one instruction after the other. We really haven't yet got to the point where CAD system or CASE-type tools help out very much. In order to move toward what we call higher levels of software automation in the future, we are going to be using more standardized systematic-type modules for developing software systems.

The end aim of automatic programming is a complete system without the need to write any code. At present you don't see automatic programming, where you simply say to the computer: *"OK, I need a program to do this, and lo and behold, out it comes"*.

This paper describes aspects of applied research related to the development of an intelligent system *DESTA*. The idea is very simple: we must get lots of different software from lots of different places that must work together and must talk to each other and the output of one can be used as input of the other. Obviously we need to have vast knowledge base for it. The software should be developed from *reusable* software components: ***"software chips"***. To do it we need to consider some general issue:

- Knowledge content, structure, representation, acquisition, and maintenance.
- Inference engine.
- Human-computer interaction, natural language interface, integrated program development environment.

2. Knowledge Base

Software development is an intensely knowledgebased activity. The functioning or activity of any intelligent system (natural or artificial) can be reduced to solving a set of suitable problems, 93% of which belong to so-called "ill-defined" problems whose solution cannot be expressed by formulae or by means of using classical or modern mathematics. In this case it is more convenient for the end user to specify their requirement to the computer by means of natural language (NL).

By an *intelligent system* we shall understand a system which enables us to solve *intelligent problems*.

2.1. Intelligent Problem

Intuitively under the problem **T** they will understand the four **<X,Q,F,Y>**, in which **X** stands for the finite set of input data and their specification; **Q** represents the goal descriptions, and **F** is the finite sequence (or *set*) of rules transforming **X** into **Y**. Thus **Y** is the finite set of output data.

Proceeding from a given definition it is easy to single out at least three classes of problems, which are characterized by the following relations:

X & Q & F |-- Y, (1)
X & Q & Y |-- F, (2)
X & Q |-- F => Y, (3)

where symbols "**&**", "**|--**" and "**=>**" stand for "**and**", "**give**" and "**implication**" respectively.

Intelligent problems are characterized by relations **(2)** and **(3)**. In this case the problem to define **software chips** can be represented by:

T = <Sp(x),Dt(x),Nm(F),As(F),Sp(F),Cnd(F),Dc(F),Pr(F),Sp(y),Dt(y)>,

where **Dt(x)** and **Dt(y)** are the *"input"* and *"output"* data, respectively;

Sp(x) and **Sp(y)** are their *"specification"*;

Nm(F) is the *"algorithm name"* coinciding with the problem name **Nm(T);**

Dc(F) represents the *"declarative description"* of the finite sequence (or set) of rules called the "**algorithm**". Having available **Dc(F)** the system **Knows How** to solve the problem **T**, but it **Cannot** (is not able to) solve this problem;

Pr(F) is the *"program representation"* of **F** called "**program**" (or "**module**"). Having available only **Pr(F)** the system **Does Not Know How** to solve the problem **T**, i.e. it cannot describe declaratively the course of its solution, but because the description of **Pr(F)** is *"intelligible"* to the system it **Can Execute** F (i.e. **Can Solve** the problem);

The description of *"functionality"* - **As(F)**, the *"condition"* of its execution - **Cnd(F)** and its *"specification"* - **Sp(F)** including the language for the description of **F**, the method of solving, the required computational resources for its implementation etc are brought to conformity with every **F**.

2.2. Content and Structure of KB

The activity of any natural (or artificial) intelligent system is just connected with solving different problems. Hence, knowledge of these systems must be **predisposed** to realize such activity. In our opinion the KB should consist of three components:

(1) Knows WHAT,
(2) Knows HOW,

(3) CAN DO Something.

According to the traditional approach to knowledge representation, the knowledge is divided into *"Declarative"* and *"Procedural"* that does not permit one to realize the main capability of the human mind: ***"bootstrapping principle"***.

One can distribute among the three part of KB the elements of problem notion:

(1) Knows WHAT: ***Sp, Dt, Nm, As, Cnd.***

Here all the declarative components connected with the specification of the problem being solved are interlinked.

(2) Knows HOW: ***Dc(F),***

(3) CAN DO Something: ***Pr(F).***

At present there can be no doubt that the possibilities of the artificial intelligence system (AIS) are defined to a large degree by the organization of the knowledge store. In the general case, under the **memory organization** one should understand the regularity of data distribution in memory assuring the storage of various links between separate elements of information and representing the main principle of gestaltpsychology, i.e. **"the whole is greater than the sum of the parts"**. In other words, the structure obtained as a result of integration should contain more information than had been used for its creation. Apparently this defines the striking ability of a human being for generating and understanding an endless number of sentences based on the limited experience with a limited number of sentences.

Moreover, at every moment of time both a man and AIS deals only with relatively small fragments of the external world. The corresponding structures are needed to integrate these fragments separated in time into the integral picture.

All kinds of binary relations can be divided into just four classes: ***"one-to-one", "one-to-many", "many-to-one"*** and ***"many-to-many"***. For implementation of these classes of binary relations the four types of elements corresponding to them are suggested: **I-elements, λ-elements, Y-elements** and **X-elements.** Different combination of these elements determine the different attributes of the structures. The KB of *DESTA*-system are provided by the interaction of the different structures as follows:

- **L-tree-structure** (combination of **I-** and **λ-elements**). This is an initial structure which provides the recognition of new words, the normalization of well-known words and the determination of direct links with the corresponding nodes of **Sm-structure, Md-structure** and **Set-structure**.

- **TB-structure** (combination of **I-, Y-** and **λ-elements**). Provides the understanding of new words.

- **Sm-structure** (combination of **I-, Y-, λ-** and **X-elements**). Provides the handling of new or well-known sentences or sequences of words.

- **Md-structure** (combination of **I-**, **Y-** and **λ-elements**). Provides the mapping of the **In-, Out-parameters** and **Cnd(F)** interaction for different modules and algorithms.

- **Set-structure** (combination of **I-**, **λ-** and **Y-elements**). Usually each module is associated with several other modules logically including it or included by it. This structure permits the system to map such links.

- **PrRI-structure** (combination of **I-** and **λ-elements**). Provides the storing of production rules and direct access to them.

3. Inference Engine

In the past AIS development was based on the *Logical Paradigm,* the main idea of which was to extract the problem solving from some theorem proof using, for example, first order predicate calculus (or Horn clause logic which is the restriction form of first order predicate calculus).

At present it is understandable that the KB of the real AIS is ***incomplete, inconsistent*** and *should be* ***open.*** In such a case it would be natural to devote more attention to the *human inference process* which is based on *"plausible reasoning"* using maximum "argumentation" about problem solving within the framework of KB.

One can single out at least three relatively independent mechanisms serving the *"natural inference"* source:

- **integration of information;**
- **addition of information;**
- **cognitive transformations.**

The idea of the *structured approach* for natural inference is considered.

4. Natural Language Interface

The natural language (NL) was chosen as:

- an *external* language for knowledge descriptions;
- an *internal* language for knowledge representation;
- a *specification* language of the problem being solved and non-procedural or procedural algorithm;
- a *communication* language between the end users and *DESTA* system.

Any NL-text handling is performed as follows:

- Any NL-sentence is divided into so-called **"nuclear"** (the simplest) sentences. By *nuclear sentence* (NS) is meant:

- a simple or a simple extended sentence with the direct order of words, where the subject group can be expressed only by a noun.

- a simple or a simple extended sentence with the direct order of words, where the subject group is expressed by a verb in the form of the imperative mood.

• Any NS is transformed into a form of the **NL-statement.** The verb of any NS is just a name of **active** or **state** statement. *Active* NS looks like a module description (e.g. ***Remove****(what,from,to)*) and *state* NS - as a specifier (e.g. ***Be****(what,where)*). Every NL-statement consists of the operation name plus respective parameters determined by the valence of this operation or its management model or its role frames.

NL-statements represents **A-statements, S-statements** or **R-statements. A-statement** is the statement of action or the active NL-statement. **S-statement** is a state NL-statement or a statement of relations with a subject of inactive state. **R-statement** represents some relations like *syno-nym, antonym, a part of* and so on (e.g. *"Delete a cursor is a synonym to erase a cursor", "Ascending sort algorithm as opposed to descending sort algorithm", "TB-structure is a part of DESTA knowledge base").*

There is a semantic equality between the initial sentence and the finite set of NL-statements.

• The finite set of NL-statements for initial NL-sentence is represented as a ***"concept"***.

5. Conclusion

At present *DESTA* is software implemented to proof the correctness of a paradigm being suggested. In short the keystone of this paradigm is as follows:

• For the end user it is more convenient to specify their requirement to the computer by means of **natural language.**

• The system has to automatically yield a readily comprehensible good structured rapid prototype which *in all stages of structured growth* ***should be executable.***

• Software development is an intensely **knowledgebased activity.** Using NL-specification we can include in the KB functional descriptions of **"software chips"** (SC): *modules, procedures, functions, batch files, operating system commands* and *algorithms.* The SC can be implemented on any programming languages: *module-oriented, object-oriented* and/or *active declarative* languages. The NL-specifications allow us to join them in whole software systems (we are not discussing here about compatibility SC for different programming languages).

• Using NL-specification DESTA either extracts from the KB the suitable SC in accordance with the NL-specification or asks the user for a more detailed description of the problem being solved.

Generic Domain Models in Software Engineering

Neil Maiden

Department of Business Computing
City University
London EC1V OHB, UK.
Tel: +44-71-253-4399 x3422
E-mail: cc559@city.ac.uk

Abstract

This paper outlines three research directions related to domain-specific software development: (i) reuse of generic models for domain-specific software development; (ii) empirical evidence to determine these generic models, namely elicitation of mental knowledge schema possessed by expert software developers, and; (iii) exploitation of generic domain models to assist modelling of specific applications. It focuses on knowledge acquisition for domain-specific software development, with emphasis on tool support for the most important phases of software development.

Introduction

Domain-specific software design has aroused considerable interest over the last decade. Most of the research effort has focused on supporting the latter stages of software development, typified by program transformational techniques and systems (e.g. Feather 1987). However, it is now agreed that most costly problems occur during the early stages of system development, when systems' requirements are ill-defined and poorly understood. Therefore, domain-specific software development (as opposed to design) must provide effective guidance during requirements engineering and high-level software design as well as during system implementation. Unfortunately requirements engineering differs from system design in its focus on the identification and embedding of systems in their environment rather than prescribing systems' functionality. This broad view can often preclude the complete capture of all domain knowledge, implying only partial automation of domain-specific software development. This paper proposes, as a first research direction, that it is more beneficial to model generic domain models rather than specific application domains, and to exploit these generic models for guiding rather than automating requirements engineering and high-level software design.

Domain modelling is needed for domain-specific software development. However, case histories of successful domain modelling and effective methods for modelling complex applications have been lacking in the literature. Innovative work by Neighbors (1980) indicated that domain analysis was both difficult and time-consuming, even for experienced analysts. Recent findings have supported this view, for instance Prieto-Diaz (1991) reports difficulties in maintaining a domain model represented as a faceted classification scheme supporting reuse within a single application. Furthermore, models of specific applications can only support development within that application, while many organisations develop software for many applications, thus reducing the potential payoff from such application modelling. Generic domain models provide an alternative domain knowledge source which can provide greater payoff to software developers because of their applicability to many applications. Reuse of such models has been proposed elsewhere (e.g. Reubenstein & Waters 1991), although little is known about the nature, contents and applicability of generic domain models for effective requirements engineering. As a result, a second research direction proposed in this paper is to determine the knowledge structures of generic domain models which support effective requirements engineering.

Generic domain models have been proposed to support requirements engineering activities, however they may also provide effective guidance for longer-term domain modelling activities. The problem is akin to knowledge acquisition during knowledge-based system (KBS) development. Recent advances in knowledge acquisition techniques promote reuse of generic, partial domain models as templates supporting top-down knowledge acquisition and modelling (e.g. Wielinga et al. 1991,

Chandrasekaran 1986). A third research direction proposed in this paper is to exploit generic domain models to assist application modelling within a comprehensive domain modelling framework.

The remainder of the paper investigates these three research directions, namely reusing generic models for domain-specific software development, determining the nature of these generic models from empirical studies, and exploiting generic domain models to assist subsequent modelling of specific applications.

Evidence for Generic Domain Models

Evidence for the likelihood of generic domain models to assist requirements engineering comes for current software engineering research, recent advances in knowledge acquisition and empirical evidence of software engineering expertise. Each is examined in turn.

Generic Domain Models in Software Engineering

Generic domain modelling in software engineering research has arisen as an issue in both automated software development and domain analysis. Reusable generic domain models have been proposed in several research projects (e.g. Reubenstein & Waters 1991). The well-known Requirements Apprentice (Reubenstein & Waters 1991) exploits cliches representing general software engineering concepts, including domains, however few clues are provided about the nature and boundaries of these cliches. Furthermore object-oriented paradigms have been limited to design and implementation phases of software development while object-oriented analysis has focused on object definition rather than object structure within domains. This would suggest that abstraction in software engineering is poorly understood, and requires further investigation.

Iscoe (1991) reviewed evolving research in domain modelling, with emphasise on meta-models instantiated into application domains. His research issues include domain classification and analysis, implying the need for a theory of software engineering abstraction, however he gives few clues about the nature of this abstraction. Several domain meta-models have been reported in the literature (e.g. Lubars 1988, Dardenne et al. 1991, Chung et al. 1990), however this work has not been sufficiently developed as application examples and in practice to determine generic domains. Prieto-Diaz (1990) also reviewed domain analysis and emphasised the importance of abstraction in domain modelling. However, he could offer no guidance for this abstraction process beyond current structured analytic techniques such as SSA (De Marco 1978) and domain analyst expertise. Furthermore abstraction was limited to identification of important domain features rather than generification from application instances.

Generic Knowledge Structures in Knowledge Acquisition

Knowledge acquisition techniques and methods (reviewed in Neale 1988) have implications for domain analysis for at least three reasons. First the task of requirements analysis is similar to knowledge acquisition. Aspects of KBS development such as information analysis, application selection, project management, user requirement capture, modular design and reusability are similar to those encountered in software development. Indeed the KADS project (Wielinga et al. 1991) proposes a sequential development method based on modelling activity and an operational model that exhibits some desired behaviour in terms of real-world phenomena, similar to many existing software development methodologies including SSADM (Cutts 1987) and JSD (Jackson 1983). A second reason is that knowledge acquisition techniques like KADS are relevant to requirements engineering because they focus support on the earlier, analytic stages of KBS development while domain-specific software design paradigms support later stages such as program specification, transformation and maintenance (e.g. Feather 1987). Finally knowledge acquisition approaches introduce techniques not found in otherwise equivalent software development methodologies, so a review of knowledge acquisition techniques in respect to requirements engineering is warranted. The following knowledge acquisition projects were identified as having implications for generic domain models.

Generic Tasks: Chandrasekaran and his colleagues at Ohio State University propose generic tasks to provide an outline or framework for expert system design. This framework claims that complex knowledge-based reasoning tasks can often be decomposed into generic tasks, each with associated types of knowledge and family of control regimes (Chandrasekaran 1986). Six generic expert system tasks are identified in terms of knowledge types and control regimes: classification, state abstraction, knowledge-directed retrieval, object synthesis by plan selection and refinement, hypothesis matching, and assembly of compound hypotheses for abduction. These tasks encompass both declarative and procedural knowledge in reoccurring patterns. They emphasise the importance of domain knowledge and the reuse of large knowledge structures akin to complex objects.

The KADS Project: KADS is an ESPRIT project (ESPRIT-1 P1098), providing the knowledge engineer with reusable partial knowledge models as templates to support top-down knowledge acquisition and modelling, based on recognition that parts of the model are not specific to certain applications. The success of this approach has been documented in many domains, including diagnosis of

movement disorders, paint selection, commercial wine making and statistical consultancy (Wielinga et al. 1991), suggesting the potential effectiveness of the retrieval and exploitation of generic knowledge structures in complex, ill-structured modelling activity. Generic models are categorised by system structure, solution type and the discrepancy between observed and expected behaviour, based on a modified and extended version of Clancey's (1985) description of problem types.

KADS's domain meta-model is based on a tentative topology of primitive problem solving actions, or knowledge sources, consisting of concepts, their attributes, the values of these attributes, the structure of concepts, sets and set instances. It is derived from the type of operation that is carried out by the knowledge source, demonstrating the importance of contextuality linked to functionality of knowledge needs. KADSs' generic models demonstrate the importance of a topology of primitive problem solving actions based on a taxonomy of problem solving types. This approach has lead to considerable modelling success in a number of complex applications. Unfortunately the meta-model is weak due to the varied nature of domains tackled by the KADS approach.

Generic Mechanisms: Klinker et al.'s generic mechanisms (1991) result from comprehensive research to develop constructs which are both usable and reusable during knowledge acquisition and modelling. These mechanisms represent generic tasks reoccurring in many domains, for example *sizing* and *scheduling* tasks occur in both the computer and aerospace industries. Klinker's current knowledge acquisition tool is populated with at least 14 such mechanisms which are also aggregated into larger applications in which they often occur. A theory of mechanisms is currently being developed from experiences with the knowledge acquisition tool in new applications, leading to a more refined and complete mechanism library. The approach of Klinker and his colleagues differs from those of Chandrasekaran and KADS in terms of the research methods used, which employ empirical evidence to determine generic task mechanisms and their aggregation. This most comprehensive generic domain analysis demonstrates the importance of multi-level abstraction and granularity for generic domain models, with a need to aggregate domain models in several dimensions such as common application groupings.

Summary: Recent knowledge acquisition approaches demonstrate the feasibility of guidance based on generic domain and task models during complex modelling activities like requirements engineering. However, a model of generic tasks and domains, implying an underlying theory of abstraction, is not readily available for software engineering researchers. Such a theory must identify several determinants of generic domain models, such as their appropriate level of abstraction, granularity and effective knowledge structures, to decide how big or small these generic domain models should be. Intermediate findings point to potential research directions, namely the contextual nature of these models and the need to validate them through empirical evidence in software engineering, for instance software engineering domains are very different to those of commercial winemaking or diagnosis of movement disorders (Wielinga et al. 1991).

Software Engineers' Expertise

Software engineers' expertise offers one form of empirical evidence for validating generic domain models. Expert software developers possess preformed abstract mental schema of domains which allow them to classify, structure and scope each problem (Guindon 1990) and develop multiple mental domain models (Pennington 1987). Experts' mental schemata can be assumed to be effective generic representations due to successive refinement during requirements engineering experiences in many applications, which may suggest why experienced software engineers are much sought-after individuals. Intelligent software development mimicking experts' knowledge structures may be one direction for research to proceed. Again however, current empirical evidence of software engineers' mental schema is limited due to a lack of relevant and comprehensive studies, so more effective, empirical research is needed to determine generic domain models in software engineering.

An Initial Model of Software Engineering Abstraction

Studies of generic models in software engineering, knowledge acquisition and expert analytic behaviour suggest the validity of a generic domain modelling approach to domain-specific software development. However, the nature of these generic models is less clear, so a three-phase research strategy was adopted at City University to determine their contents and structure:

- investigation of analogical specification reuse as one means of determining generic domains underlying this reuse, to be followed by validation and extension of these generic domain models using:
- empirical studies of software engineers' mental knowledge structures via knowledge acquisition techniques, and
- domain analyses of large, real-world applications to verify generic domains in terms of recognisable instantiations and instantiation aggregations.

The first phase is partially complete while the second and third phases are the focus of an ESPRIT Basic Research Action. The first phase has led to a tentative model of software engineering domains which provide the basis for

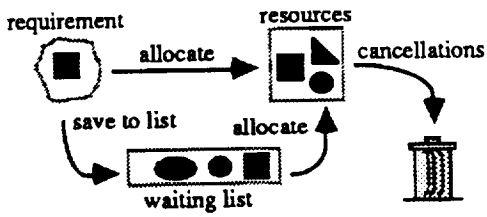
a retrieval mechanisms supporting analogical specification reuse, and described in Maiden & Sutcliffe (1991).

Generic Domain Models Supporting Specification Reuse

Maiden (1991) identified an initial model of generic domain models through studies of analogical specification reuse, such that two specified domains are analogous if they are both instances of the same generic domain class, as demonstrated in Figure 1. As such the scope, granularity and level of abstraction of these generic knowledge structures is constrained to most effectively support reuse of functional specifications.

Maiden's model (1991) proposes that generic domain classes are differentiated by key state transitions, hence a generic resource hiring domain, of which library loans is an example, can be distinguished from a generic resource containment domain (e.g. stock control) by the key transition of return (see Figure 2). Similarly two classes of object allocation domain can be differentiated by the transitions *send to* and *remove* from waiting lists, for example the reservation system of a local cinema may not include waiting lists once all seats for a performance are sold. Additional determinants of distinct domain classes were identified in terms of these critical transitions between domain states. The following meta-schema for describing critical generic domains and their instantiations was developed, with each knowledge type describing one or more critical dimensions:

- actions leading to state transitions with respect to a knowledge structure. These actions represent system intervention in the domain to maintain or change the domain from a possible to a required state. Actions and state transitions are central to the model, for example the allocation action in the theatre reservation example causes the object (*theatregoer booking*) to change state from an in-requirement state to an occupying-resource state (from *required-booking* to *reserved-booking*);
- object structural knowledge describing both problem and required domain states in the form of conceptual relations between objects. For example, *theatre contains many seats*, each *containing one or no theatregoer booking*. Furthermore, required knowledge structures such as *maximise seat occupation*, can be imposed on these domain states;
- pre/post-conditions on state transitions identified from values describing the current state of objects, for example a state transition moving the theatre reservation to the seat only occurs if the reservation and the seat have similar constraints such as *non-smoking*, *price <£20*, *seat is unreserved*, etc.;
- object types describe object roles in the context of state transitions, for example customer bookings is a type of *requirement* while theatre seats are *resources* available to satisfy those *requirements*;
- functional transformations which may be causally-related to state transitions in the domain model, for example the functional transformation *allocate from waiting list* results in a state transition moving the theatre booking from the waiting list to theatre seats while functional transformations in library systems are typically *lend* and *return*;
- state transitions can also be distinguished by their triggering events. Domain events which cause state transitions are either initiated by the information system or by events external to it, for example the theatre reservation domain may in part be distinguished by the scope of triggering domain events because allocating customer bookings to the seats available is initiated by the information system while removing customers from allocated seats results from external events.



<world,reqt_set,has_one>
<world, resource_set,has_one>
<world,list,has_one>
<reqt_set,reqt,contains_one>
<resource_set,resource,has_many>
<resource,reqt,contains_one>
<allocate,reqt,reqt_set,resource,o ne>
<reqt,reqt_type>
<resource,resource_type>
<allocate,matching_properties>

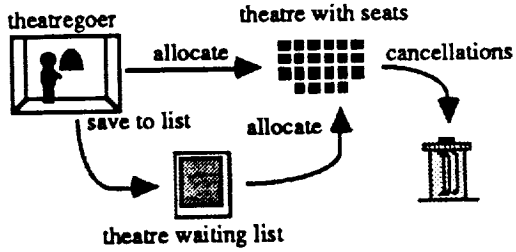


Figure 1: simple theatre reservati on domain and its generic domain class, incl uding partial definition of that class

To sum, this model of generic software engineering domains was developed from example-based studies of such domains in the context of reuse. Its development was driven by domain-based studies of important knowledge structures in software engineering, a constraint which

distinguishes it from existing meta-models of software engineering domains such as TELOS, (Chung et al. 1990). The extent and nature of this example-driven analysis is described briefly in the following section.

Example Generic Domain Models

Current research has identified 35 generic domain models through the relatively weak proof of trial by example, see Figure 2 and Maiden (1992). These models were hierarchically-structured to identify classification and specialisation of basic domain types, for instance library and stock control domains are both specialisations of a more generic object containment domain. Furthermore generic domains were aggregated to identify *standard* applications incorporating many domain classes in unique patterns, for example a comprehensive library system can involve lending, stock updating, allocating and reserving activities which are all instantiations of different domain classes. The validity of this current approach is suggested by a prototype specification reuse tool incorporating 10 such generic domain models in a specialisation hierarchy to support successful retrieval and explanation (Maiden 1992). However, further work is needed to extend and validate the current model.

Domain Modelling From Generic Domain Models

This paper reports studies which reveal domain analysis to be a problematic task akin to knowledge acquisition. Parallel experiences in knowledge acquisition suggest that generic domain models may assist in this task. A domain modelling framework incorporating reuse, similar to the KADS method, is needed to make effective use of generic domain models. In particular such models provide pieces of the generic skeleton to be instantiated and fleshed out with additional knowledge types until the domain model is complete.

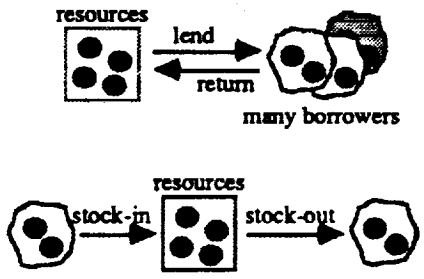


Figure 2: examples of generic doma in models:
(i) renewable resource, e.g. libra ry,
(ii) non-renewable resource, e.g. stock control.

Summary

This paper proposes that greater benefits can be achieved from modelling generic domains rather than specific applications, so overcoming domain modelling bottlenecks by mimicking expert software engineering practice. Intelligent tool support founded on generic domain knowledge can assist during requirements engineering in the following tasks:

- identification and validation of application models to assist effective requirements capture, providing intelligent feedback on system requirements and models;
- procedural guidance for requirements engineering tasks, using generic domain hierarchies to focus on critical domain features and incrementally specialise them;
- support for reuse through categorisation of problems based on generic domain classes (Maiden & Sutcliffe 1991).

We would also intuitively expect generic domain models to provide the basic building blocks for complex application modelling then domain-specific software design. Acquiring these knowledge structures therefore takes on considerable importance for intelligent support during requirements engineering and software design. To this end we suggest that much research effort should be focused on practical and empirical research to determining the most effective knowledge structures for supporting domain-specific software development.

References

Chandrasekaran B., 1986, Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design, *IEEE Expert* 1(3), 23-30.

Chung L., Katalagarianos P., Marakakis M., Mertikas M., Mylopoulos J. & Vassilou Y., 1990, From Information System Requirements to Designs: A Mapping Framework, Technical Report CSRI-245, University of Toronto, September 1990.

Clancey W.J., 1985, Heuristic Classification, *Artificial Intelligence* 27, 289-350.

Cutts G., 1987, *SSADM - Structured Systems Analysis and Design Methodology*, Paradigm Publishing.

Dardenne A., Fickas S. & Lamsweerde A., 1991, Goal-directed Concept Acquisition in Requirements Elicitation, Proceedings of 6th Intl Workshop on Software Specification and Design, Como (It) 25-26th October 1991, IEEE Computer Society Press, 14-21.

De Marco T., 1978, *Structured Systems Analysis and Specification*, Prentice-Hall International.

Feather M.S., 1987, A Survey and Classification of some Program Transformation Approaches and Techniques, *Program Specification and Transformation*, ed. L.G.L.T. Meertens, Elsevier Science Publishers.

Guindon R., 1990, Designing the Design Process: Exploiting Opportunistic Thoughts, *Human-Computer*

Interaction 5, 305-344.

Iscoe N., 1991, Domain Modelling: Evolving Research, Proceedings of 6th Knowledge-Based Software Engineering Conference', Syracuse NY, 22-25th September 1991, 300-304.

Klinker G., Bhola C., Dallemagne G., Marques D. & McDermott J., 1991, 'Usable and Reusable Programming Constructs', *Knowledge Acquisition* 3, 117-135.

Lubars M.D., 1988, A Domain Modelling Representation, MCC Technical Report STP-366-88, Software Technology Program, MCC, Austin Texas, November 1988.

Maiden N.A.M., 1992, Analogical Specification Reuse during Requirements Analysis, PhD Thesis, Department of Business Computing, City University.

Maiden N.A.M., 1991, Analogy as a Paradigm for Specification Reuse, *Software Engineering Journal* 6(1), 3-15.

Maiden N.A.M & Sutcliffe A.G., 1991, Analogical Matching for Specification Retrieval, Proceedings of 6th Knowledge-Based Software Engineering Conference, Syracuse NY, 22-25th September 1991, 101-112.

Neale I., 1988, First Generation Expert Systems: A Review of Knowledge Acquisition Methodologies, *The Knowledge Engineering Review* 2, 105-145

Neighbors J.M., 1980, Software Construction using Components, Ph.D. Dissertation, Department of Information and Computer Science, University of California, Irvine.

Pennington N., 1987, Comprehension Strategies in Programming, *2nd Workshop of Empirical Studies of Programmers*, ed. G. Olson, S. Sheppard and E. Soloway, Ablex, 100 - 113.

Prieto-Diaz R., 1991, 'Implementing Faceted Classification for Software Reuse', *Communications of the ACM* 34(5), 88-97.

Prieto-Diaz R., 1990, Domain Analysis: An Introduction, *ACM SIGSOFT Software Engineering Notes* 15(2), April 1990, 47-54.

Reubenstein H.B. & Waters R.C., 1991, 'The Requirements Apprentice: Automated Assistance for Requirements Acquisition', *IEEE Transactions on Software Engineering* 17(3), 226-240.

Wielinga B.J., Schreiber A.Th. & Breuker J.A., 1991, 'KADS: A Modelling Approach to Knowledge Engineering', Technical Report ESPRIT Project P5248 KADS-II, May 1991.

DOMAIN-SPECIFIC FUNCTIONAL SOFTWARE TESTING: A PROGRESS REPORT

Uwe Nonnenmann
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
un@research.att.com

1 Introduction

Software Engineering is a knowledge intensive activity that involves defining, designing, developing, and maintaining software systems. In order to build effective systems to support Software Engeneering activities, Artificial Intelligence techniques are needed. The application of Artificial Intelligence technology to Software Engineering is called Knowledge-based Software Engineering (KBSE) [Lowry & Duran, 1989]. The goal of KBSE is to change the software life cycle such that software maintenance and evolution occur by modifying the specifications and then rederiving the implementation rather than by directly modifying the implementation. The use of domain knowledge in developing KBSE systems is crucial.

Our work is mainly related to one area of KBSE that is called automatic specification acquisition. One example is the WATSON prototype [Kelly & Nonnenmann, 1991] on which our current work is based. WATSON is an automatic programming system for formalizing specifications for telephone switching software mainly restricted to POTS, *i.e., plain old telephone service.*

Other examples of such systems are IDeA and Ozym. The Intelligent Design Aid (IDeA) [Lubars & Harandi, 1987] performs knowledge-based refinement of specifications and design. IDeA gives incremental feedback on completeness and consistency using domain-specific abstract design schemas. The idea behind Ozym [Iscoe *et al.*, 1989] is to specify and implement applications programs for non-programmers and non-domain-experts by modeling domain knowledge.

However, despite two decades of moderately successful research, there have been few practical demonstrations of the utility of Artificial Intelligence techniques to support Software Engineering activities [Barstow, 1987] other than such prototypes as mentioned above. Our current approach differentiates itself from these other approaches in two antagonistic ways: On the one hand, we address a large and complex real-world problem instead of a "toy domain" as in many research prototypes. On the other hand, to allow such scaling, we had to relax the ambitious goal of complete *automatic programming*, to the easier task of *automatic testing.*

2 KITSS Overview

In the *Knowledge-Based Interactive Test Script System (KITSS)*, we have taken this philosophy and applied it to the task of functional software testing. In functional testing, the internal design and structure of the program are ignored. It corresponds directly to uncovering discrepancies in the program's behavior as viewed from the outside world. This type of testing has been called *black box* testing because, like a black box in hardware, one is only interested in how the input relates to the output. The resulting tests are then executed in a simulated customer environment which corresponds to verifying that the system fulfills its intended purpose.

Tests are by definition correct but not exhaustive. KITSS checks and augments given tests and generates related new ones but does not generate the full specification as in WATSON. KITSS can be seen as performing testing from examples. KITSS' strength lies in its very *domain-specific* approach [Barstow, 1985] and customized reasoning procedures. It will change the software life cycle by modifying the functional tests and then rederiving the system tests which corresponds to finding and eliminating software problems early in the development process as in the KBSE paradigm. Therefore, we designed KITSS to be well integrated into our existing design and development process [Nonnenmann & Eddy, 1991].

KITSS achieves this integration by using the same expressive and unobtrusive input medium, namely *test cases.* They describe in English the high-level details of the external design and are written before coding begins. KITSS also produces the same output as before, executable *test scripts* written in an in-house test automation language. These are low-level descriptions derived from test cases for specific test equipment.

To support this integration, KITSS has a *natural language processor* that is trained in the domain's technical dialect [Jones & Eisner, 1992] and converts the

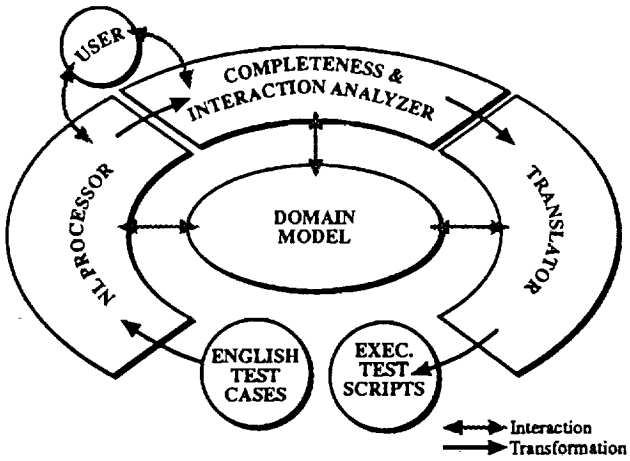


Figure 1: KITSS Architecture

test cases into a formal representation that is audited for coverage and sanity. To accomplish this, KITSS uses a customized theorem prover-based *analyzer* (based on WATSON technology) and a hybrid knowledge base as the *domain model* using both a static terminological logic and a dynamic temporal logic. These two modules have been feasible only due to the domain-specific knowledge-based approach taken in KITSS. Finally, a *translator* takes the corrected test case and converts it from temporal logic into a test script language that can exercise the switch using dedicated test equipment. Figure 1 shows the overall architecture of KITSS.

In summary, KITSS helps the test process by generating more tests of better quality and by allowing more frequent regression testing through automation. Furthermore, tests are generated earlier, *i.e.*, *during* the development phase not *after*, which should detect problems earlier, thus resulting in reduced maintenance costs (for more details on KITSS see [Nonnenmann & Eddy, 1992]).

3 Knowledge Representation Issues

As we used a highly domain-specific approach, the domain model is one of the center pieces of KITSS. In the following section we will highlight the key design decisions made and the knowledge representations chosen.

Testing is a very knowledge intensive task. It involves experience with the switch hardware and testing equipment as well as an understanding of the switch software with its several hundred features and many more interactions. There are many binders of feature descriptions for PBX software, but no concise formalizations of the domain were available before KITSS. The focus of KITSS and the domain model is on an end-user's point of view, *i.e.*, on (physical and software) objects that the user can manipulate. Figure 2 gives an overview of KITSS' domain model.

The *static model* represents all telephony objects, data, and conditions that do not have a temporal extent but may have states or histories. It describes major hardware components, processes, logical resources, the current test setup, the dial plan and the current feature assignments. All static parts of the domain model are implemented in CLASSIC [Brachman *et al.*, 1990], which belongs to the class of terminological logics (*e.g.* KL-ONE).

The *dynamic model* defines the dynamic aspects of the switch behavior. These are constraints that have to be fulfilled during testing as well as the predicates they are defined upon. Objects include predicates, stimuli which can be either primitive or abstract, and observables. Additionally, the dynamic model includes invariants and rules as integrity constraints. Invariants are assertions which describe only a single state, but are true in all states. These are among the most important pieces of domain knowledge as they describe basic telephony behavior as well as the *look & feel* of the switch. Rules describe low-level behavior in telephony. This is mostly signaling behavior.

Representing the dynamic model we required expressive power beyond CLASSIC or terminological logics, which are not well-suited for representing plan-like knowledge. We therefore used the WATSON Theorem Prover, a linear-time first-order resolution theorem prover with a weak temporal logic. This non-standard logic has five modal operators *holds, occurs, issues, begins,* and *ends*. As an action *occurs*, the response to that action may endure until some other action occurs or it may be transient. An enduring actions *begins* and *holds* until, in response to some other action, it *ends*. In the transient case, the switch merely *issues* the response. These modals are sufficient to represent all temporal aspects of our domain. The theorem proving is only tractable due to the tight integration between knowledge representation and reasoning.

In adding the dynamic model, we were able to increase the expressive power of our domain model and to increase the reasoning capabilities as well. The integration of the hybrid pieces did produce some problems, for example, deciding which components belonged in which piece. However, this decision was facilitated because of our design choice to represent all dynamic aspects of the system in our temporal logic and to keep everything else in CLASSIC.

The domain model consists of over 600 domain concepts, over 1,700 domain individuals, and more than 160 temporal axioms.

The domain model was built in the initial phase of the project as the reasoning modules depended on the underlying representations being created first. In this phase the domain model changed constantly as we still enhanced our understanding of the domain. Then, we left the domain model mainly unchanged through the development phases until a milestone was reached. We

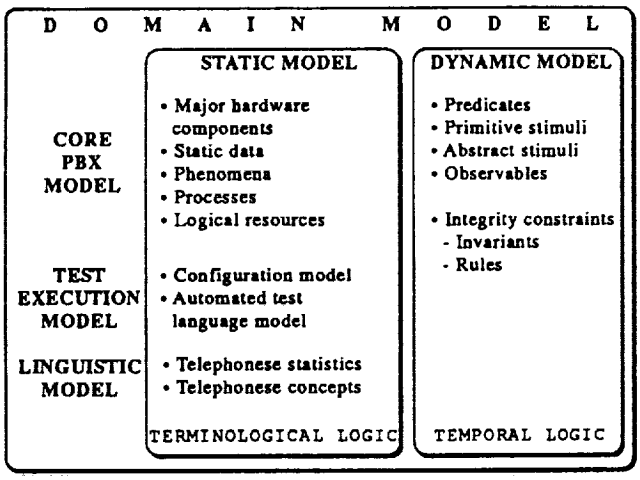


Figure 2: KITSS Domain Model

then typically performed a major revision of the domain model based on problems encountered. The domain model went through three iterations like that and has been stable since. Of course, we continually add new knowledge but the representations are mainly unchanged.

Although we anticipate that the domain model will grow only linear with the number of features covered, we already had great difficulty in acquiring new knowledge and maintaining the existing domain model as both tasks have been done completely manual so far. Therefore, knowledge acquisition and maintenance support is crucial. At least we have gained an understanding on how to design such automated tools. The above experience is the main motivation for another approach included as a proposal in this workshop's proceedings [Hall, 1992].

4 Status

At last year's ASD workshop, we initially reported on KITSS. Since then, we had a major evaluation of the KITSS prototype with the following results.

System execution speed has not been a bottleneck due to continued specialization of the inference capability. However, it is not clear how long such optimizations can avoid potential intractability of the theorem prover. Another result we found was that it is easier to train the natural language processor than it is to achieve coherence in the reasoning audits. The initial scaling phase from the proof-of-principle demo (3 test cases) to the first prototype (38 test cases) was successful but took too long and KITSS was too brittle in general. Although KITSS has been designed from the beginning as an interactive system (the "I" in KITSS), we pushed the *machine initiative* in this scaling phase as far as possible as an experiment. However, this was the limiting factor for rapid progress.

The current schedule is to expand KITSS to cover a few hundred test cases in the next couple of months. To achieve such scaling, we changed some of KITSS' design to make it much more robust. Despite the fact that the natural language processor performed well, we augmented it with a paraphrasing mechanism, *i.e.*, in case the English input cannot be understood, the user can rephrase this input using a paraphrase language based on our temporal logic. When the analyzer encounters problems, it intensely questions the user to explain unclear passages of test cases. This approach has only been possible because KITSS is very responsive. Additionally, we changed all reasoning modules to produce "soft-failures", *i.e.*, in case the system fails to fully understand the test case it still continues to translate user inputs as an *assistant*.

In general, our strategy has shifted from a nearly fully automatic system to one that is much more interactive and might resort to the assistant paradigm occasionally. However, we still have the full KITSS approach in place so that we can improve *e.g.* the analyzer incrementally without any change for the user other than reduced interactions.

5 From Research Into Practice

In 1987, the initial prototype of WATSON was completed. Today, five years later, KITSS is a prototype with a more restricted scope in the same domain although with broader coverage. So where is the big progress? Or in other words: How do we get from a research prototype to a deployed real-world system? Is this "just" technology transfer? Another question might also be: Can we build a research prototype in a toy domain and expect it to work on a real-world problem?

The answers to these questions[1] are not easy, but we would like to give at least partial answers based on our experience.

In scaling from WATSON's toy domain to KITSS' real-world telephony domain we had to address a number of new research issues (which we can just list here without further explanation). For example, we had to extend our temporal logic, restructure the domain model into a hybrid one (static/dynamic), create telephony "micromodels" and understand their interactions, reason with multiple agents instead of single ones, significantly enhance the planning component, understand the "purpose" of inputs to be able to generalize and specialize them, perform more complex nonmonotonic reasoning etc. Additionally, we had to incorporate and customize a natural language module to cover input test cases in English instead of a limited scenario language.

KITSS being so different in all these respects, WATSON cannot be seen as a core system to which we just

[1] These questions are based on a personal discussion with Ron Brachman.

added domain knowledge. For KITSS, we had to basically rewrite and enhance WATSON and add completely new modules. We see KITSS based on technology that WATSON has proven feasible. Therefore, we do not see KITSS as technology transfer at all but as a research project that covers a real-world domain.

Yet despite such research progress, we were still required to further change KITSS' design to achieve practical solutions (see Section 4). No matter how impressive a research prototype looks, we believe there is still a lot of research to be done in order to scale to real-world use.

So: "Can we build a research prototype in a toy domain and expect it to work on a real-world problem?" The answer is: "Probably not".

6 Conclusions

KITSS is a domain-specific system to generate executable functional tests using the KBSE paradigm. Its main difference to previous approaches is that it covers a real-world domain instead of a toy domain. However, therefore the initial goal of automatic programming had to be limited to the easier problem of automatic testing.

KITSS converts input tests into a formal representation interactively with the user. To achieve this, we needed to augment the static domain model represented in a terminological logic with a dynamic model written in a temporal logic. Knowledge acquisition has been performed manually and is a major problem that has not been addressed yet.

In general, scaling from a research prototype to a real-world system involves much additional research before the actual technology transfer can begin. To achieve such scaling, we had to further move toward more user interaction. Although scaling-up remains a hard task, KITSS demonstrates that our KBSE approach chosen for this complex application is feasible.

Acknowledgments

Many thanks go to John Eddy, Van Kelly, Mark Jones, and Bob Hall who also contributed major parts of the KITSS system. Additionally, we would like to thank Ron Brachman for his support throughout the project.

References

Barstow, D.R.: Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, November 1985.

Barstow, D.R.: Artificial Intelligence and Software Engineering. In *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, 1987.

Brachman, R.J., McGuinness, D.L., Patel-Schneider, P.F., Alperin Resnick, L., and Borgida, A.: Living with CLASSIC: When and how to use a KL-ONE-like language. In *Formal Aspects of Semantic Networks*, J. Sowa, ed., Morgan Kaufmann, 1990.

Hall, R.J.: Interactive specification acquisition via senarios: A proposal. In *Proceedings of the AAAI'92 Workshop on Automating Software Design*, San Jose, CA, 1992.

Iscoe, N., Browne, J.C., and Werth, J.: An object-oriented approach to program specification and generation. *Technical Report, Dept. of Computer Science, University of Texas at Austin*, 1989.

Jones, M.A., and Eisner, J.: A probabilistic parser applied to software testing documents. In *Proceedings of the 10th National Conference on Artificial Intelligence*, San Jose, CA, 1992.

Kelly, V.E., and Nonnenmann, U.: Reducing the complexity of formal specification acquisition. In *Automating Software Design*, M. Lowry and R. McCartney, eds., MIT Press, 1991.

Lowry, M., Duran, R.: Knowledge-based Software Engeneering. In *Handbook of Artificial Intelligence, Vol. IV, Chapter XX*, Addison Wesley, 1989.

Lubars, M.D., and Harandi, M.T.: Knowledge-based software design using design schemas. In *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, 1987.

Nonnenmann, U., and Eddy J.K.: KITSS - Toward software design and testing integration. In *Automating Software Design: Interactive Design - Workshop Notes from the 9th AAAI*, L. Johnson, ed., USC/ISI Technical Report RS-91-287, 1991.

Nonnenmann, U., and Eddy, J.K.: KITSS - A functional software testing system using a hybrid domain model. In *Proceedings of the 8th Conference on Artificial Intelligence for Applications*, Monterey, CA, 1992.

A Domain-Specific Design Architecture for Composite Material Design and Aircraft part Redesign

W. F. Punch III[1], K.J. Keller, W. Bond and J. Sticklen[2]

1 Polymer Composite Materials

Advanced composites have been targeted as a "leapfrog" technology that would provide a unique global competitive position for U.S. industry. Composites are unique in the requirements for an integrated approach to designing, manufacturing, and marketing of products developed utilizing the new materials of construction. Numerous studies extending across the entire economic spectrum of the United States from aerospace to military to durable goods have identified composites as a "key" technology.

A typical chronology for designing a composite material is as follows [5]. First, macroscopic properties which are desired in the completed composite are set. Properties such as final material tensile modulus, resistance to acids and alkalis, and electrical resistance are parameterized. Based on these desired properties, the composite designer proposes an initial plan for the production of the composite. This plan includes both an ingredients list for all materials to be initially present, and a preliminary protocol which states how the initial mixture is to be processed. Next, the composite designer estimates how well the proposed composite design meets the initially stated, desired properties. This estimate is carried out along two paths. The composite designer may actually produce samples of the composite, then perform laboratory testing to determine properties of interest. Or the designer may model the proposed composite to estimate its properties. Ultimately, a proposed composite design will result in an actual material which can be subjected to laboratory testing. But, one goal of composite researchers is to provide better models for proposed designs in order to limit the number of candidate materials which must actually be fabricated for testing. Following one round of design proposing, and matching to specifications, successive rounds of redesign are usually required before convergence of proposed composite properties to desired properties takes place.

In general there have been two approaches to composite construction: build models of a given composite material, then determine characteristics of the material via numerical simulation and empirical testing a(leaders along this path include the research groups of Dr. Larry Drzal and Dr. Martin Hawley, Michigan State University Composites Center), nd experience-directed construction of fabrication plans for building composites with given properties (e.g., the recent work of Frances Abrams at AFWAL Materials Lab at Wright Patterson). The first route set a goal to capture basic understanding of a device (the composite) by use of a rigorous mathematical model; the second attempts to capture the expertise about the process of fabricating a composite (to date) at a surface level typically expressed in a rule based system. The first important point is that, from an AI perspective, these two research lines are attacking distinctly different problems. Secondly, both tracks have current limitations. The mathematical modeling approach has yielded a wealth of data but a large number of simplifying assumptions are needed to make numerical simulation tractable. Likewise, although surface level expertise about how to build a particular composite may yield important results, recent trends in the KBS area are towards augmenting surface level problem solving with deeper level knowledge.

1.1 Redesign of Existing Metal Parts from Composites

Utilizing composite parts in engineered devices offers multiple advantages over the use of traditional metal parts. These include weight savings, strength::weight ratio increase, and greater flexibility in processing. Because of these relative advantages, there is great interest in retrofitting existing metal components with composite material counterparts. However, although such retrofitting is under way, many of the advantages of composite materials are not being fully exploited. Current practice throughout many industries is to take an existing metal part, and try to redesign it piece wise as a set of very simple composite material

[1] Author to whom correspondence should be sent.

[2] Punch and Sticklen are with the AI/KBS Lab., Computer Science Department, Michigan State University, East Lansing, MI 48824. They can be contacted at punch@cps.msu.edu and sticklen@cps.msu.edu, respectively. Keller is with the AI Center, Bond is with the McDonnell Douglas Research lab, of the McDonnell Aircraft Company St. Louis, MO 63166-2995

components. From a design perspective, given present state of the art, this is a reasonable practice: the larger the composite part, and the more geometrically complex, the more difficult the design task becomes.

Many of the relative advantages of composites; e.g., the strength::weight ratio; is most prominent when the entire component is designed as a unitary piece. The bottleneck in undertaking such unitary design lies in the difficulty of the re-design task. Designing the fabrication protocols for a complex-shaped, thick section composite are currently very difficult. It is in fact, this difficulty that our research will address.

2 The Design Problem

In redesigning an existing aircraft part with a new composite part, the composite engineer is faced with numerous design and manufacturing options with interrelated effects on all aspects of the product lifecycle (strength, weight, producibility, survivability, maintainability, and supportability are some of the issues). Capturing and managing these interrelationships is necessary for successful implementation of an optimum design environment.

This process requires the integration of knowledge and data from many disciplines residing on multiple platforms and repositories. Most design decisions have interrelated effects which are extremely difficult to predict. For instance, a limited knowledge of composite materials could result in a design that will not meet temperature requirements, is improperly layed-up, will not conform to the manufacturing process, or results in a high scrappage rate when manufactured. Redesign of an existing part is more complex since it also requires the designer to reconstitute an understanding of the functions of the original part.

An intelligent design system which could integrate and arbitrate knowledge and calculations obtained from multiple sources would be a valuable resource to both the inexperienced design engineer who must design "simple" parts and also to the experienced engineer who is focusing on more complex designs. This system must be flexible and expandable to accommodate advances in composite material design and manufacturing methods and to facilitate the incorporation of knowledge and calculations from multiple sources. We plan to meet these challenges by developing a framework which can bring appropriate problem solving techniques to bear at the correct time.

2.1 Testbed Design Complexity

Our initial works concentrates on "simple" composite parts. The processes required to produce these parts are generally well known and proven. In addition, the parts that fall into this category are numerous. Simple parts include clips, brackets, avionic shelves, doors, etc. Complex parts such as outer moldline skins and internal bulkheads, frames, and longerons typically account for only 10 percent of the number of parts that go into an airframe.

The current design process is outlined in Figure 1 which describes the inputs, the design process, and the desired outputs.

To automate these tasks, the overall system architecture shown in Figure 2 is anticipated.

2.2 Testbed Design Architecture

The system will provide support to the design engineer in a number of different ways:

- Interactive design advice: will check input design parameters and part data for errors and design rule violations. If the feature violates a design rule, the system informs the designer.
- Process and material selection: provide the user with predictions of the cost and probability to successfully produce a given part and/or engineering feature. These predictions can be based on knowledge-based results or on analytical process simulation models.
- Analysis support: provides a method to obtain strength information based on closed form solutions. This would permit a designer to perform quick strength predictions before carrying out complete finite element analysis.

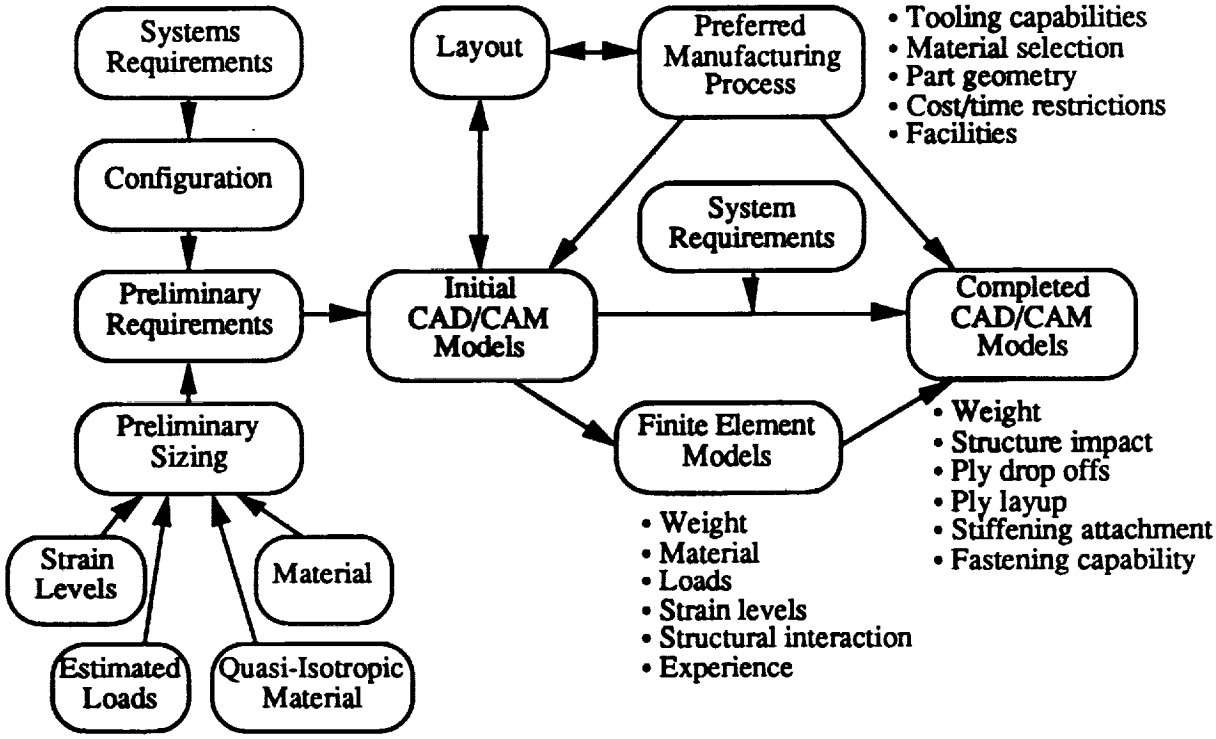


Figure 1: The composite design process

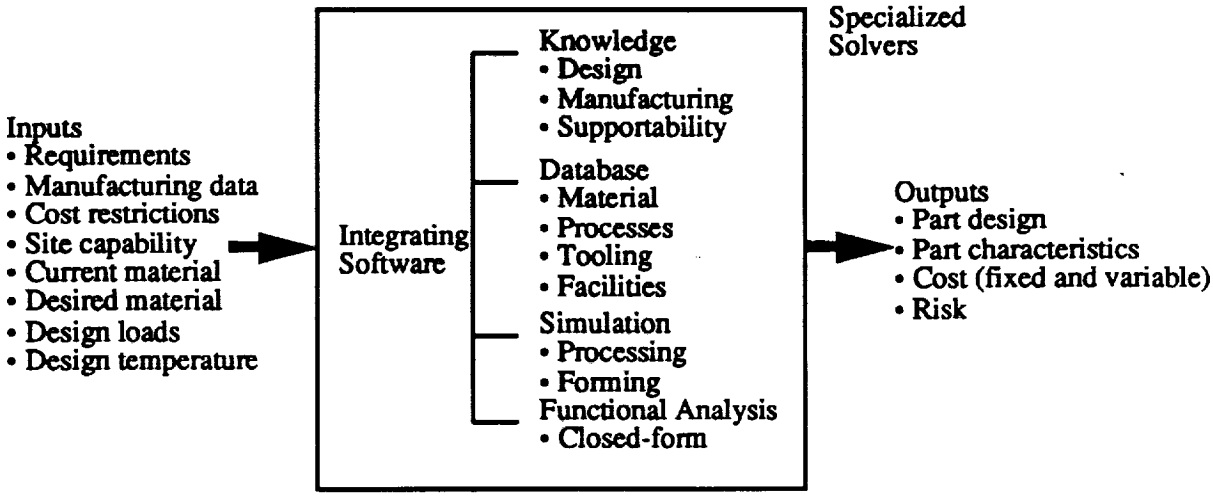


Figure 2: The expected system architecture

- Process plan support: produce process documentation by completing standard templates based on design and manufacturing rules (a good example is a ply orientation table based on standard design rules for given material and strength requirements)

3 Role of AI in the Composite Materials Redesign Domain

In the composite materials redesign area, the heterogeneous problem solving techniques which we elaborated above are all required to produce robust problem solving results. In addition, we are investigating the integration of other, more well explored problem solving methods which include:

- *design problem solving:* _ Design problem solving will play an important role in our application domain. We intend to utilize Routine Design (Brown & Chandrasekaran, 1986) as a starting point - similar to the applications of Sticklen et. al. (Sticklen,Kamel,Hawley, & Delong, 1991). In addition, we may utilize a case base of known designs which must be altered in minor ways to meet requirements - much in the spirit of the work of Goel on redesign using a functionally indexed case base (Goel & Chandrasekaran, 1989).

- *capturing design rationale:* _ As pointed out above, current practice for the problem of redesign of metal parts from composite materials is hampered by concentrating on small scale replace, rather than entire component redesign. A major reason for this is that the purposes of the component to be redesigned are not available. We will meet this problem by first capturing the component purpose utilizing a *Functional Reasoning* (FR) Approach - following the work of Bond and Sticklen in the aerospace domain (Bond,Sticklen, & Pegah, 1991; Pegah,Bond, & Sticklen, 1991; Sticklen,Bond, & St. Clair, 1988).

- *simulation in the service of design verification:*_ Once a composite materials-based redesign for a component is proposed, there is a need to "test" it. This will be undertaken using a combination of FR+bond graph simulation. A functional representation augmented with primitives in bond graph will form the basis for a simulation of the *process* of composite material curing. This simulation will yield the characteristics of the cured composite produced by following the proposed design.

4 Integration Architecture

A pivotal part of our research lies in the integration architecture which will allow the smooth interaction of both disparate problem solving agents, and will allow the access and use of heterogeneous knowledge and data bases. For our initial exportation, we will apply the Task Integrated Problem Solver (TIPS) approach [3, 4, 2]. Characteristics of TIPS which we are important for domain-specific design are:

1. TIPS provides dynamic integration. That is, the which problem-solver is invoked is determined by the problem state, previous problem-solving history, knowledge available and other factors. The "chunk" size of the methods that TIPS supports is higher than than of SOAR and this is helpful for applications where much knowledge is compiled with nevertheless need for runtime integration.

2. TIPS exploits the task structure, ie. the goal-subgoal structure of the overall problem, to identify methods that might be relevant.

3. TIPS can mix different types of problems solvers. The problem-solvers need some minimum of communication capabilities, however, but this can be added to the kernel problem solver.

4. TIPS is capable of supporting task-level explanation of why a particular problem-solver was invoked.

The basis for the representation of control used in TIPS is the Sponsor-Selector system first used in DSPL (Design Specialists and Plans Language) [1]. It consists of a hierarchy of three parts: at the top a *selector*, under the selector some number of *sponsors*, and under each sponsor a *method invocation*. In short, the available problem-solving methods are grouped under the selector as sponsor-method pairs, where

each sponsor provides appropriateness measures for its associated method invocation. At any control choice point (i.e., some point in the flow of problem-solving at which another method could be invoked) the overall control process is to run all the sponsors to rate their associated methods, then have the selector choose the next method to be executed based on the sponsor values and other data.

The sponsors are therefore used as "local" measures of how appropriate a problem-solver is for achieving the current goal, while the selector takes a more "global" view of selecting which of the available methods is the "best" under the present circumstances. Both the sponsors and selectors encode their knowledge in a patter-match table that indicates what features are important for making the decision and how those combinations of features contribute to the final answer (selection or appropriateness measure).

The TIPS architecture has been used to implement a large-grained medical diagnosis system in the domain of liver and blood disorders [4] integrating Compiled/Association-Based Diagnosis, Causal Reasoning, Data Gathering, Data Validation, Therapy Planning and User Interaction.

4.1 Integrated Reasoning in the Composite Domain

Consider the problem as presented by a portion of Figure 1. **Configuration** and **Preliminary Sizing** are two methods whose results can be used by the method **Preliminary Requirements** to set up "rough design" parameters in the early stage of the design. In fact, the representation as listed could be more complicated, **Preliminary Requirements** might require an number of invocations of both the **Configuration** and **Preliminary Sizing** methods to reach a stable configuration where each invocation requests small modifications to the initial answers. Each new invocation would contain information about the new problem that **Preliminary Requirements** has perceived, perhaps even some suggestions about how to repair the problem, and asks for further refinement. Repeated invocations continues until the **Preliminary Requirements** method has achieved its goal, and problem-solving then continues to more detailed designing. The *dynamic* interaction between multiple methods as shown in this example is the kind of problem-solving that an integrated reasoner has to capture to be effective in this domain. That is, determining when a method is done depends on the state of the present problem and the perception by the system of which goals are "active" and if they have been achieved.

5 Future Directions, Extending the Integration Architecture

The first problem is one of representation. Figure 1 is a direct representation of the methods used to achieve goals in the composite design problem, but not the goals implicit in guiding the problem-solving. In the current TIPS implementation, the methods are directly represented, but the goals are only represented implicitly in the sponsor-selector system.

The second problem is one of standardizing the means by which sponsors can monitor goal status and by which methods can indicate their success, partial success or failure. Likewise, a common means by which problem state information is gathered must be made available. At present, Lisp code specific to the goals and methods in the diagnostic system have been used but this needs to be supported by an architectural feature.

Both are important additions to a TIPS architecture. Direct representations of the goals of a domain make the job of mapping an analysis directly to code much simpler. It also enhances other aspects of a system, such as explaining why certain steps were taken in a case run (because the goal situation at that stage was X and Y was the best choice etc.). Standardized interactions between methods is also quite important will enable cooperation and negotiation among problem-solving methods to solver larger problems. These problems and others will form the basis for research on domain-specific design in the years to come.

References

[1] D. Brown and B. Chandrasekaran. *Design Problem Solving: Knowledge Structures and Control Strategies.* Pitman, London, UK, 1989.

[2] W. F. Punch III. *A Diagnosis System Using a Task Integrated Problem Solving Architecture (TIPS), Including Causal Reasoning.* PhD thesis, The Ohio State University, 1989.

[3] W. F. Punch III. TIPS (task-integrated problem solver), a task-specific integration architecture for heterogeneous agents. In *Proceedings of the AAAI-91 Workshop on Cooperation Among Heterogeneous Intelligent Systems*, pages 1–10, Anahiem, CA, July 1991.

[4] W. F. Punch and B. Chandrasekaran. An investigation of the roles of problem-solving methods in diagnosis. In *Proceedings of Second Generation Expert System's Conference*, pages 25–36, May 1990.

[5] V. Venkatasubramanian, Y. Lee, and C. Gryte. Design of polymer composites: A knowledge-based framework. In *AIChE87*, 1987.

RT-Syn: A Real-Time Software System Generator

Dorothy E. Setliff
Electrical Engineering Department
University of Pittsburgh
Pittsburgh, PA 15261

Abstract

This paper presents research into providing highly reusable and maintainable components by using automatic software synthesis techniques. This proposal uses domain knowledge combined with automatic software synthesis techniques to engineer large-scale mission-critical real-time software. The hypothesis centers on a software synthesis architecture that specifically incorporates application-specific (in this case real-time) knowledge. This architecture synthesizes complex system software to meet a behavioral specification and external interaction design constraints. Some examples of these external constraints are communication protocols, precisions, timing and space limitations. The incorporation of application-specific knowledge facilitates the generation of mathematical software metrics which are used to narrow the design space, thereby making software synthesis tractable. Success has the potential to dramatically reduce mission-critical system life-cycle costs not only by reducing development time, but more importantly facilitating maintenance, modifications and extensions of complex mission-critical software systems which are currently dominating life-cycle costs.

1 Introduction

The software development process is time consuming, expensive, and fraught with errors. These characteristics hinder the reuse of software. This paper presents an approach that seeks to reduce software development time while simultaneously increasing software reuse. This approach, called RT-Syn, uses characteristics of the software domain to synthesize software. Although there is a wealth of structural and syntactic knowledge that can be brought to bear for software synthesis, the lack of success in generalized software synthesis [15] argues that this knowledge is insufficient, and that domain-specific knowledge is necessary for successful software synthesis.

The chosen domain is real-time software. Software for real-time applications can be characterized as a set of time-constrained tasks. Real-time system failure is defined as when any task misses its hard deadline. Software development and subsequent redevelopment is one of the major bottlenecks of real-time system design and maintenance. One method to remove this bottleneck is to automatically synthesize real-time software to meet all system platform and task timing requirements. We argue that the presence of strict operation requirements, such as task-level timing constraints and compiler and target platform constraints, can be used to guide synthesis.

RT-Syn builds off of previous work by Setliff [11, 12], Barstow [1], and Kant [9] among others. Generality is supported by the use of a visual language as an algorithm specification. Only that application-specific knowledge relevant to programming-in-the-small synthesis is currently incorporated within the RT-Syn synthesis architecture. Within the vernacular of this particular application domain, this current project focuses on task-level synthesis issues. Future work encompasses system-level (programming-in-the-large) synthesis issues.

Section 2 reviews the current RT-Syn 1.0 architecture and presents a brief overview of each component within RT-Syn 1.0, including an enumeration of what knowledge is required to perform task-level synthesis and how that knowledge is represented. Section 3 describes RT-Syn 2.0, an approach to system-level software synthesis. This section describes the various components envisioned within RT-Syn 2.0 and their interactions. Finally, Section 4 presents an overview of our progress and a plan for future work.

2 RT-Syn 1.0 Architecture

This section describes the architecture of our initial version of RT-Syn called RT-Syn 1.0. This version incorporates knowledge of the effects of a specific platform, the DLX processor [4], on task-level synthesis, as well as an algorithm that uses this knowledge to formulate timing and space requirements estimates for different implementation possibilities for a task. We chose the simulated DLX processor for this initial version because it is completely modelled and DLXsim (a program which models the operation of the DLX processor) keeps statistics useful for checking our design decisions. Current work in predictable platforms [18] seeks to design and develop real-world platforms which exhibit the similar predictable characteristics exhibited by the DLX processor.

RT-Syn 1.0 integrates platform characteristics to synthesize a real-time software task to meet hard deadline requirements. The RT-Syn 1.0 automatic synthesis system has five key features. First, a visual graphical user interface, called Intuition, captures application algorithms without implementation specifications. Second, RT-Syn 1.0 analyzes the task-level data and control flows to produce worst-case timing

and space predictions. Third, RT-Syn 1.0 uses these predictions to select abstract representations of data structures and algorithm implementations to meet required timing and space constraints. Fourth, RT-Syn 1.0 synthesizes C code from the selected implementations. Fifth, RT-Syn 1.0 validates the execution of the code to meet the predicted timing and space utilization values.

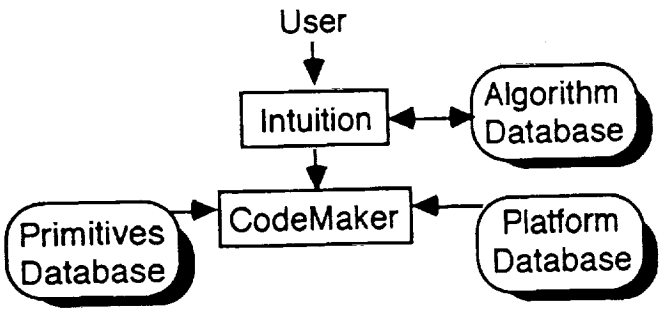


Figure 1: RT-Syn 1.0 Synthesis Architecture

RT-Syn 1.0 (See Figure 1) is composed of several databases and two tools: Intuition, a graphical programming language, and CodeMaker, a real-time software directed program synthesis system.

A visual programming language is a way to program a system using a visual paradigm. Visual programming languages vary in the level of represented detail [5]. Intuition provides a semantic-level, user-friendly, hierarchical visual programming environment. This environment offers several advantages over text-based environments. It allows for the user-friendly input of RT-Syn specifications with minimum effort. Its hierarchical nature allows the user to easily modify the description. There is also less learning time involved. The concepts behind these advantages are present in a variety of existing visual programming research efforts [2, 3, 10, 13, 14].

Intuition acts as a user input system as well as a high level visual algorithm description. Intuition descriptions capture both data flow and control flow information. Intuition is similar in form and function to the popular object-oriented drawing program that exists on personal computers, such as MacDraw. The user has a palette of tools, and a window in which to draw a schematic representation of algorithms. Figure 2 is an example of a Intuition screen representation of a FFT signal processing algorithm. Each rectangular cell on the screen represents a basic building block of the algorithm. The building block may be direct computation or a hierarchical reference to another Intuition file representing quantities of computations. Data and control flow is captured by the connecting lines between the cells. There is no data

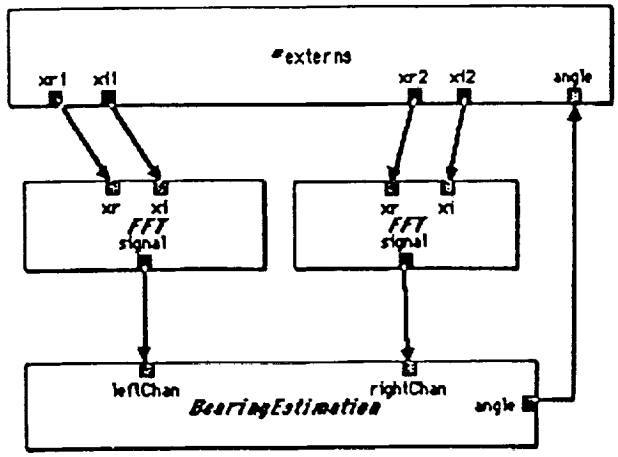


Figure 2: Representative Intuition Algorithm Representation

typing or language representation implied within Intuition. Groups of Intuition algorithmic representation formulate the Algorithm Database. This database organizes the algorithms by function and by hierarchical components.

The cornerstone of the RT-Syn 1.0 schema is a program synthesis system which is capable of producing extremely well-modelled executable code. CodeMaker, a program synthesis system targeting real-time signal processing software, guarantees the validity of the generated code using the following methodology. First, CodeMaker analyzes all algorithmic representations available within Intuition that can result in the required behavior and produces an internal representation of the algorithmic data and control flow. Second, CodeMaker uses knowledge of the platform to produce implementation ranges for each specifiable resource (currently, speed and space). All implementation possibilities for a particular task are guaranteed to be within the implementation range. Third, CodeMaker analyzes the implementation ranges to select the algorithmic approach most likely to satisfy all of the required specifiable resource quantities for a particular task. Fourth, CodeMaker then synthesizes an implementation of the algorithm that is guaranteed to meet the required specifications.

The following example illustrates the operations within CodeMaker, what knowledge is applied, and how knowledge is applied. In this example, CodeMaker is directed (via Intuition) to synthesize a FFT algorithm that executes in X cycles and may consume no more than Y bytes of data memory. There exist numerous algorithms that exemplify the FFT behav-

ior [6]. Each algorithm, regardless of the implementation, places a differing strain on computation and memory resources. CodeMaker must select a particular algorithm and then synthesize an implementation of the algorithm to specifically meet the requirements for that task.

CodeMaker first analyzes all algorithmic possibilities and produces a control and data flow graph for each. The implementation ranges are formed by noting a fairly simple fact. Timing limitations generally adversely affect space utilization (less time requires greater space requirements). To discover the minimum speed characteristic for any algorithm, synthesize the algorithm while attempting to reach a 'time = 0' constraint. Of course, this is impossible so synthesis will fail, but the resultant implementation specification will be the closest to zero and thus the minimum time and maximum data memory. Setting a 'space = 0' constraint and synthesizing to meet that constraint find the implementation that uses the least data memory and maximum time. Program memory requirements are not considered currently in CodeMaker. Each algorithm that satisfied the required behavior is synthesized twice to produced implementation ranges for that algorithm. The synthesis process incorporates knowledge of the platform and maintains any dependencies within an algorithm. This approach is used to prune out those possibilities that will fail early on in the process. The algorithm with a design space closest to the resource requirements is selected. It is important to note that the design space is not convex. There exist points in the design space that are not reachable. The synthesis process is capable of backtracking back to this point if the requirements are not attainable during the final synthesis process.

Synthesis of an implementation requires access to specific platform-dependent data. The Platform Database contains modelling information needed to generate accurate predictions about the timing and space utilization characteristics of tasks for a given platform. The bulk of the information in this database consists of data about primitives. Primitives are the lowest-level building blocks used by RT-Syn when synthesizing C code. Typically, primitives represent single C operations, such as addition, multiplication, and branching operations. Information stored in the platform database includes: the timing and space characteristics of all primitives on this platform, formula to account of eccentricities in the behavior of the model of the platform, characterization of any operating system calls that will be used, and a characterization of the C compiler to be used when generating executable code. The behavior of platforms and compilers vary widely. As a basic example, some computers utilize a separate floating-point math unit, which makes floating-point math a faster alternative to integer math. At a compiler level, different compilers perform different optimization techniques, so that the same piece of C code compiled on two different compilers and then run on the same platform will have different characteristics. An important feature of the RT-Syn system that is a result of the Platform Database is that a given set of tasks can be completely resynthesized for different platforms by changing only the platform selection.

CodeMaker uses the data and control flow graphs, in tandem with knowledge of the tradeoffs requisite in a particular target platform (e.g., operating system, hardware) and target language primitives, to select implementations. The selection process is performed bottom-up (or correspondingly output-back-to-input) on the data flow graph. This graph walking approach specifically acknowledges the impact of external requirements (the required outputs) on the implementation selections. In this way, the external requirements are applied as early as possible and are used to reject portions of the design space that are unworkable. Only that knowledge pertinent to the task-level prediction and selection is required. Analysis, prediction, then selection continues until the task implementation is fully specified. At this point, the implementation code is constructed.

Experimentation using Intuition and CodeMaker [16] demonstrate both the efficient synthesis (100's of lines of code is less than 30 seconds on a MAC II) possible when application-specific knowledge is incorporated within synthesis. Numerous experimentation [16, 17] also shows the range of implementations attainable merely by modifying the task-level constraints. These experiments show the power of using application-specific knowledge to synthesize software to meet a set of specifications and thereby provide for software reusability. Only knowledge of platforms and primitives particular to data structure and algorithm selection is incorporated and used within CodeMaker. This knowledge prunes the design space, while providing solutions for time and space constrained signal processing tasks.

RT-Syn 1.0, a real-time task set synthesis architecture, synthesizes viable C code solutions based on a user's high-level task set specification. RT-Syn 1.0 takes as input a high-level description of the real-time tasks to be generated, along with information about the equipment on which the system will be run and generates code to implement each task. The code is guaranteed to meet any resource use requirements. The next section describes work in progress towards system-level synthesis using the successes of the RT-Syn 1.0 system as a foundation.

3 Towards System-Level Synthesis

RT-Syn 2.0 focuses on system-level synthesis. RT-Syn 2.0 performs all of the synthesis provided by RT-Syn 1.0 and also generates a scheduler to coordinate the execution of the individual tasks. The entire system will be guaranteed to produce the desired outputs within the given deadlines and without exceeding the host equipment's resources.

The RT-Syn 2.0 real-time software synthesis architecture is shown in Figure 3. Each block in the diagram represents a component in the synthesis architecture. There are two types of components in the synthesis architecture: design and database components. We first introduce each component, then describe the advantages of this architecture. We then describe in detail the functionality of each component

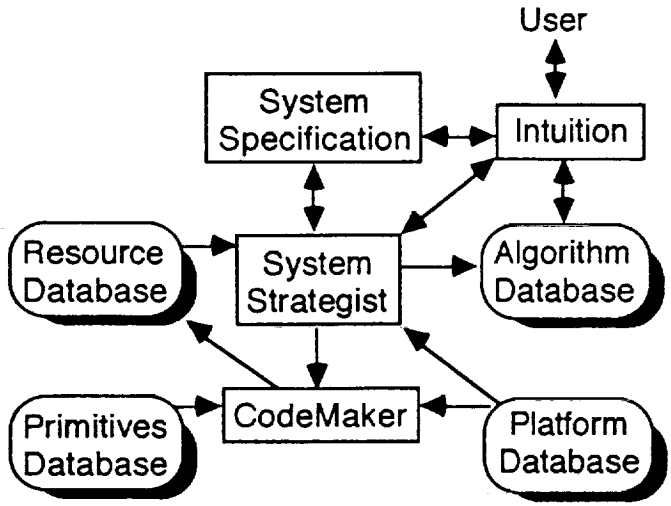


Figure 3: RT-Syn 2.0 Synthesis System

in the order that synthesis follows: from user specification of a set of tasks to the successful synthesis of code for each task. The software synthesis process is not monolithic; rather, the synthesis process is composed of a succession of abstraction levels. Breaking the software synthesis architecture into the various abstraction levels illustrates the design hierarchy. Each design abstraction level is composed of a design component with database components as required. The highest abstraction level is the user interaction with the synthesis architecture. The System Specification component interacts with the user to form a system-level behavior description. The user interacts with the synthesis architecture via Intuition.

The second abstraction level decomposes the system behavior descriptions into task level specifications. The System Strategist component analyzes the system-level behavior description, determines what tasks are required, how the tasks are to be scheduled, and how resources are to be allocated to each task. All of these operations require knowledge. This knowledge is captured within three database components: the Resource Management Database, the Algorithm Database, and the Platform Database. The Resource Management Database contains knowledge of current system resource allocations and results from prior software synthesis operations. This information is used within the System Strategist to aid the design and synthesis process. The Algorithm Database contains knowledge about a variety of useful algorithms and the methods in which they are implemented. Each algorithm in this database is represented using Intuition. The Platform Database contains modelling information for various computer/compiler platforms. This information is used in the analysis tools within the System Strategist component and CodeMaker tool.

The third abstraction level decomposes the task level specifications into code implementations. This abstraction level has already been developed and discussed in a prior section of this proposal.

There are several advantages to this architecture. The decomposition via abstraction level provides a design focus and limits the amount of design search. The decomposition also closely mirrors the current development process of a system analyst and programmer. By mirroring this development process, it is possible to allow both system analysts and programmers to interact with the synthesis architecture. The distinction between design and database components provides a growth mechanism.

Isolating the information into three distinct databases facilitates the expansion of RT-Syn 2.0 to work with a variety of systems. Extending the knowledge of the Algorithm Database enables the system to synthesize a wider array of tasks. Adding information about more systems to the Platform Database allows RT-Syn 2.0 to model new hardware/compiler platforms. Finally, by updating the Primitives Database, RT-Syn 2.0 gains the ability to synthesize code in different languages.

The System Specification component serves three functions. The first function is to enable the user to specify the set of tasks to be synthesized. The user specifies a type of task set (out of a list of tasks of which the system has knowledge), provides information about the number of and type of inputs and outputs to the set of tasks, and chooses a target platform (from a list of machines for which there exist timing models). From this information RT-Syn 2.0 constructs a suite of tasks which will perform the desired function and run within the confines of the target platform's constraints. The output is information to the System Strategist about what tasks are required and what platform is being used. This task specification information can be very coarse-grained (e.g., system-level input) or extremely detailed (e.g., implementation-level input), depending on the level of user interaction. The second function of the System Specification component is to interact with the user during the design and synthesis process. Interaction takes the form of behavior simulation. In this way, the user can validate that the set of tasks given in the input do indeed satisfy the required mathematical functionality before synthesizing to meet the needs of the application.

The Resource Management Database provides information about system resource allocation results to the System Strategist. As individual tasks are synthesized, the amount of resources required will solidify from coarse estimates to accurate predictions. During the synthesis process, the Resource Management Database is updated to reflect the current state of a task's resource allocation needs. These needs are rep-

resented as the set of task descriptors Ci, Ti and Ui (which represent worst-case execution time, execution period, and utilization, respectively) [8, 7].

The System Strategist acts as the systems analyst for the synthesis of the set of tasks. Its initial function is to determine what task implementations will be used in the desired system and to choose and develop a real-time scheduler to manage these tasks. To choose a scheduler the System Strategist must have knowledge of a variety of scheduling schemas and have heuristics for deciding which one is most applicable to the current situation. These heuristics consist of rules derived from real-time scheduling theory. The following describes the algorithm database and provides a detailed discussion of the operations the System Strategist performs once the synthesis process is underway.

There are two main characteristics of the Algorithm Database: organization and database entry contents. The algorithms are organized hierarchically by behavior. Figure 4 illustrates the organization of the Algorithm Database:

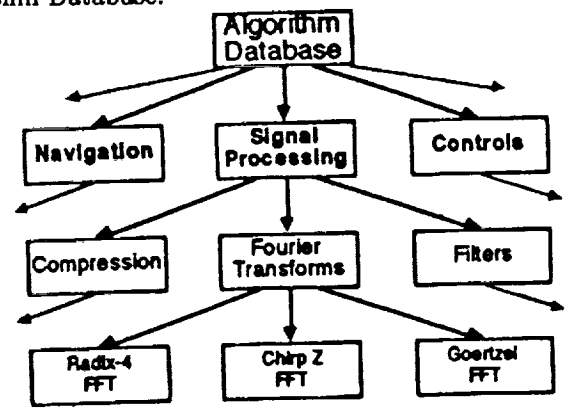


Figure 4: Design Organization of the Algorithm Database

The advantage of this organization is the reduction of search required to find all algorithms with a certain behavior. Organization by behavior places each algorithm in a specific behavior class. Each entry in a particular class possesses the identical behavior but differs in the corresponding functionality. A behavior hierarchy corresponds to the desired system-level user interaction. A typical system level specification consists of a set of behavior descriptions. These behavior descriptions match particular class and subclass behaviors in the algorithm hierarchy. All algorithms corresponding to the specified behavior may be immediately retrieved for analysis. Each entry in the database is a 3-tuple: algorithm representation, resource constraint ranges, synthesis history. Each algorithm is represented in Intuition. Each entry contains coarse-grained resource characteristics. These coarse-grained characteristics define the implementation design space for that algorithm entry. Currently the Algorithm Database contains time and space resource characteristics.

The System Strategist analyzes the system-level behavior description as a set of tasks, schedules each task, and allocates resources to each task in the system. The input to the System Strategist is a system-level behavior description. The output is a system scheduler algorithm selection, and a set of task-level descriptions. A task-level description details the behavior algorithm and desired time and space characteristics for each task. Analysis is required to synthesize the scheduler algorithm and task-level descriptions. The System Strategist first accesses the implementation range information in the Algorithm Database for each task in the system. The set of implementation ranges defines the design space that the scheduling algorithm must guarantee. There is an enumerable set of scheduling algorithms. The System Strategist attempts to synthesize all known scheduling algorithms. The System Strategist applies the underlying mathematics of each scheduling algorithm to the set of implementation ranges. A task specification is chosen within each range that will result in guaranteed schedulability. The set of task specifications may not overutilize system memory constraints. A potential scheduling algorithm is removed from consideration (pruned) when it fails to guarantee the task specification set.

Scheduling algorithm analysis iterates with the synthesis of each task in the system. Task synthesis results in more exact information than the implementation range information in the Algorithm Database. More exact information prunes the design space and allows the reallocation of resources.

4 Conclusions

The real-time software development process is time consuming. This paper presents work in progress towards alleviating the costs of real-time software system design, development, and maintenance. This work incorporates state-of-the-art scheduling theory within a software synthesis architecture. This architecture leverages off of past research into the successful synthesis of software. Results illustrating the ability to accurately predict the time and space characteristics of a task and then synthesize an implementation to meet the prediction can be found in [16, 17]. This synthesis system successfully generates functional C code from high-level algorithmic descriptions. The generated code can be modeled for speed and space requirements, and these predictions prove to be reasonably accurate for a variety of platforms. The ability to generate predictable code is the cornerstone of the RT-SYN system. By demonstrating the validity of the synthesis system, we validate the premise of automated real-time task set synthesis.

The scope of this paper presents work in the initial phase of the design and development of a real-time software synthesis architecture. We target the synthesis of uni-task systems with a focus on the use of prediction to aid synthesis. Future work in this phase encompasses expanding the repertoire of algorithms within the Algorithm Database and verifying efficient synthesis of these algorithms. The second phase of this work targets on multi-task synthesis. Specifically, we will incorporate the RT Mach operating system into the Platform Database. We see this second phase as

proving the predictability of RT Mach by accurately predicting synthesis results. The third and final phase of this work targets system-level synthesis. At this point, we will introduce real-time network characteristics into the System Synthesis schedulability analysis. These characteristics encompass adding a real-time database and system-level application scenarios.

References

[1] D. Barstow. *Automatic Program Construction Techniques*, chapter The roles of knowledge and deduction in algorithm design, pages 201–222. McMillan, 1984.

[2] S.K. Chang. *Principles of Visual Programming Systems.* Prentice Hall, Englewood Cliffs New Jersey, 1990.

[3] S.K. Chang. A visual language compiler for information retrieval by visual reasoning. *IEEE Transactions on Software Engineering*, 16:1136–1149, October 1990.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, San Mateo California, 1990.

[5] S.K. Chang T. Ichikawa and P.A. Ligomenides, editors. *Visual Languages.* Plenum Press, New York, 1986.

[6] S.M. Kay. *Model Spectral Estimation Theory and Application.* Prentice Hall Signal Processing Series, 1988.

[7] L. Sha J.P. Lehoczky and R. Rajkumar. Scheduling algorithms for real-time operating systems. Technical report, Department of Computer Science, Carnegie-Mellon University, December 1986.

[8] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20 (1):46 - 61, 1973.

[9] E. Kant F. Daube W. MacGregor and J. Wald. *Automating Software Design*, chapter 8: Scientific Programming by Automated Synthesis. AAAI Press, 1991.

[10] M. Eisenstadt J. Domingue T. Rajan and E. Motta. Visual knowledge engineering. *IEEE Transactions on Software Engineering*, 16:1164–1177, October 1990.

[11] D. Setliff and R. Rutenbar. On the feasibility of synthesizing cad software from specifications: Generating maze router tools in elf. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(6):783–801, June 1991.

[12] D. Setliff and R. Rutenbar. Knowledge representation and reasoning in a software synthesis architecture. *IEEE Transactions on Software Engineering*, Accepted for publication in special issue on Knowledge Representation to appear in September 1992.

[13] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Transactions on Computers*, 16:57–69, August 1983.

[14] D. Harel H. Lachover A. Naamad A. Pnueli M. Politi R. Sherman A. Shtull-Trauring and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16:403–414, April 1990.

[15] H.A. Simon. Whether software engineering needs to be artificially intelligent. *IEEE Transactions on Software Engineering*, SE-12(7):726–732, July 1986.

[16] T.E. Smith and D.E. Setliff. Towards an automatic synthesis system for real-time software. In *Proceedings of the 12th IEEE Real-Time Systems Symposium.* IEEE, December 1991.

[17] Tobiah E. Smith. Constraint-driven synthesis of signal processing algorithms. Master's thesis, Elec. Eng. Department, Univ. Of Pittsburgh, Pittsburgh, PA, November 1991.

[18] H. Tokuda and M. Kotera. A real-time tools set for the arts kernel. In *Proceedings of 9th Real-Time Systems Symposium.* IEEE, December 1988.

Automating FEA Programming

Naveen Sharma
Institute for Computational Mathematics
Department of Mathematics and Computer Science
Kent State University
Kent, OH 44240-0001
Email: sharma@mcs.kent.edu

Abstract

In this paper we briefly describe a combined symbolic and numeric approach for solving mathematical models on parallel computers. An experimental software system, PIER, is being developed in Common Lisp to synthesize computationally intensive and domain formulation dependent phases of FEA solution method. Quantities for domain formulation like shape functions, element stiffness matrices etc. are automatically derived using symbolic mathematical computations. The problem specific information and derived formulae are then used to generate (parallel) numerical code for FEA solution steps. A constructive approach to specify a numerical program design is taken. The code generator compiles application oriented input specifications into (parallel) f77 routines with the help of built-in knowledge of the particular problem, numerical solution methods and the target computer.

Introduction

Engineers and scientists frequently encounter mathematical models based upon partial differential equations (PDEs) in a wide variety of applications. Finite element analysis (FEA) (Zienkiewicz 1980) is a major computational tool for the numerical solution of boundary and initial value problems that arise in stress analysis, heat transfer and continuum mechanics of all kinds. The problem domain is first *discretized* into a suitable *mesh of elements*. Then well-selected analytical approximations are used for solution within each element. The global solution for all discrete points (element *nodes*) of the mesh is computed by numerical iterations taking into account inter-element interactions and boundary conditions.

Simple FEA applications can be performed with canned packages such as NFAP (Chang 1980) and NASTRAN. Situations involving complicated boundary conditions or element properties, non-linear material properties, require customizing many aspects of FEA. In such cases, the finite element solution process consists of a symbolic computation phase followed by a numerical computation phase. Depending on the problem at hand, the symbolic computation phase may involve *construction and analysis of solution approximations, simplification of large analytical expressions, changing variables and/or coordinates to simplify the problem, operating on matrices and tensors with symbolic entries*, as well as *integration and differentiation of analytical expressions*. Results of the symbolic computation phase are then used to construct numerical programs.

Frequently the mathematical models and related computer programs are revised during research, engineering and production. Numerical convergence problems may also require that a different numerical procedure be used for FEA solution steps. When the models are three-dimensional, or use large data sets, the program execution speed is critical. Writing programs for parallel computers to speed-up execution is indeed not a trivial task for modelers. Also, parallel programs written for a parallel machines can not be ported to other machines without significant re-programming effort. State of the art *parallelizing compilers* (Kuck 1978), (Allen & Kennedy 1985) take an existing (sequential) code as input and can produce programs for the target parallel machine. However, these compilers parallelize scientific and engineering applications on the model of linear algebra and either completely ignore the *domain specific* parallelism naturally present in the problem or query the user during compilation.

In recent years, there has been an increase in research and development efforts to alleviate these problems. Existing approaches combine symbolic and numerical computing in various ways. These (coupled) symbolic-numeric systems generally take the user input in a very high-level form and automatically generate numerical code in a procedural programming language like f77 or C for the target computer. Some notable recent projects are Ellpack (Rice, Boisvert, and Ronald 1985), Sinapse (Kant et al. 1990), Alpal (Cook 1990), PDEQSOL (Hirayama, Ikeda, and Sagawa, 1991), (Pe-

[0]Work reported herein has been supported in part by the Army Research Office under Grant DAAL03-91-G-0149

skin 1987), and (Steinberg and Roache 1990). Many of these projects have adopted *finite difference* solution method for PDEs.

Our Approach

We have been working for a number of years (Wang 1986), (Sharma and Wang 1988a), (Sharma 1988b), (Sharma and Wang 1990), (Sharma 1991a) in this research direction and our primary PDE solution method is FEA. We identify *key* solution steps of FEA which are **compute-intensive** and are **reprogrammed** every time new element formulations or boundary conditions are used. Our approach is to employ symbolic computation to generate sequential and parallel numerical codes for the key FEA solution steps. The code is generated in (the parallel version of) f77 on the target parallel computers (currently include Sequent Balance shared memory and distributed-memory Intel iPSC/860). Based on the user input, quantities such as **element shape functions** and **strain-displacement** matrices can be derived using symbolic mathematical computations. The derived formulas are used to generate numerical code for computing element stiffness matrix, solution of system of equations and other solution steps. The generated code can be readily combined with existing FEA codes. The overall scheme is pictorially depicted in Fig. 1. We are developing a new FEA code generator named **PIER** to build upon our previous work in this area and to break new grounds. PIER is Common Lisp (CL)-based and can work directly with the CL-based MAXIMA. It can be easily ported to other CL-based symbolic computing systems. PIER generates sequential and parallel codes for the key solution steps. In next two sections we briefly discuss our design objectives and PIER's programming knowledge. PIER input specifications and the code generation scheme are overviewed in subsequent sections. We conclude the paper by outlining some relevant issues.

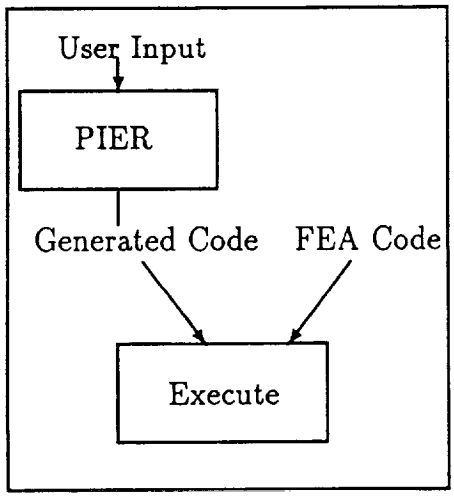


Figure 1: Overview of Approach

Design Goals

Previous FEA code generators, such as FINGER, P-FINGER and PDEQSOL, are equipped with a fixed number of numerical algorithms for FEA solution steps and the algorithms are parallelized and implemented for one specific parallel computer. Porting the code generator to other machines, thus requires work from scratch. This is a major drawback which should be overcome. Our first design requirement addresses this issue.

> The code generator provides a set of architecture-independent input specifications to design and express numerical algorithms used in FEA solution steps.

In generating parallel programs from the user input specifications, it is possible to take advantage of domain independent parallelism which exists among concurrently schedulable code modules and domain specific parallelism such as carrying out FEA (sub)computations in the *element-by-element* (Winget and Hughes 1985) formulation as opposed to the *assembled* formulation, or substructuring the FE mesh etc. This leads to our second design requirement.

> Automatically generates good implementation mappings (for input specifications) to modern high-performance computers with the help of built-in knowledge of the application domain, FEA solution method, and the target programming environment.

These design requirements allow engineers and scientists to customize the FEA solution process for the desired application area and the problem instances. Only input specifications needs to be altered without worrying about implementation details or the target architecture.

System Overview

PIER provides a knowledge-based programming environment to the modelers. The architecture of the environment comprises following components.

1. **A Programming Knowledge-Base.**
2. **A set of User Input Specifications.**
3. **Code Generator.**

The programming knowledge-base provides generic *Operations* (a set of basic linear algebra computations including matrix-vector product, vector inner product, solving triangular system of equations etc.) and domain specific Operations (a set of basic finite element analysis computations including assembling element stiffness matrices, deriving shape functions, vector preconditioning etc.). A PIER Operation has four

parts: *prologue/epilogue*, a set of *algorithm schemas*, *control dependence graph* (CDG) and the associated *cost-model*. The schemas are stated as templates written in Common Lisp, which include assignments, conventional control constructs, and array/scalar computations. The CDG represents the execution dependence among several sub-computations in the Operation and the cost-model determines the execution cost (computation and communication costs) of the Operation. PIER Operations implement the intended computations in one of the following execution styles:

(S1) `Assembled`: Execute for assembled data.

(S2) `FullyParallel`: Execute for individual element data concurrently.

(S3) `BlockParallel`: Execute for a block[1] of element data.

(S4) `Scalar`: Execute for one element data at a time.

The user input specifications provide methods to specify problem parameters, desired symbolic derivation and combine Operations to construct an FEA algorithm. The code generator generates f77 programs from the user input specifications for the target architecture. The generated code is compiled and linked on the target machine and executed. The programming knowledge-base also provides completed specifications for frequently used FEA algorithms. The user can, however, specify a new algorithm and add the same to the knowledge-base. The knowledge about programming the target parallel architecture is represented as a set of transformations. These transformations convert CDGs into equivalent f77 templates. Porting to other computers, thus, require developing the set of relevant transformations.

PIER Input Specifications

One of the major research objectives in PIER is to design a set of very high level input specifications which are used by scientists/engineers as well as system developers to describe FEA computations and problem instances. We advocate a bilingual programming style in which application oriented specifications (i.e. terminology and notations as used in standard FEA texts) can be mixed with regular f77 syntax to express an FEA algorithm. In designing the PIER input specifications we seek that the specifications should be easy to understand and easy to produce by scientists/engineers and the user specifies only the functionality desired and leaves the implementation details to PIER.

The overall approach is to add statements (which represent domain-specific computations) to f77. The set of powerful statements are primarily intended for FEA algorithms. Although this scheme could easily work in other areas of scientific computing. The input specifications support the definition of the element mesh, nodal properties, various data arrays, symbolic derivation and specification of numerical algorithms for the solution procedures. Statements defining storage strategies for FEA data arrays, high-level symbolic/numerical computations (PIER Operations), and straight-line[2] sequences of Operations (PIER *Modules*) can be intermixed with regular f77 constructs to specify a desired numerical algorithm. We now describe the underlying programming model.

[1] A block is a set of elements in which no two elements share a node

The Programming Model

In general the systematic software development process begins with informal *requirement specifications*. This is followed by one or more than one *design* phases, which define a system structure meeting requirement specifications. The design phase identifies software modules and their organization. The text book style description of numerical algorithms can be expressed at this level of abstraction with relative ease and PIER automates rest of the software development phases i.e. *detail design* and *implementation* for the algorithm. While generating parallel code, the user also specifies the resource constraints (i.e. number of maximum process/processors etc.). The input specifications are hierarchical and the user expresses numerical algorithms in a bottom-up fashion by creating abstractions of higher level in terms of lower ones. This is depicted in the Fig. 2. Let us describe each level briefly.

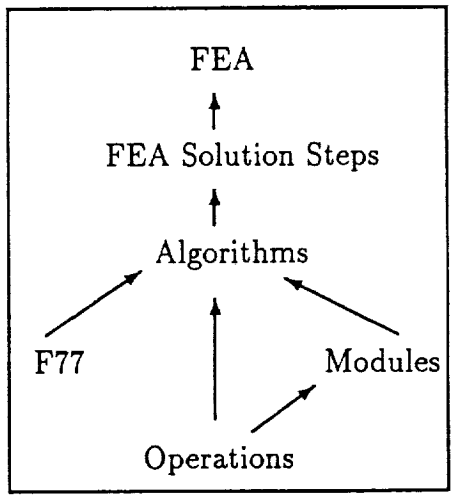


Figure 2: Hierarchy of Computations

- **Operation**: An Operation is the smallest unit of computation (provided in PIER knowledge-base). An operation usually represents a single textbook equation with one variable on the left hand side and an expression involving one or more variables on the right hand side. For example, the equations

$$Temp_1 = r \cdot z$$

[2] No sequence control is involved

$$u = a \cdot p$$

$$Temp_2 = p \cdot u$$

involved in the PCG (Preconditioned Conjugate Gradient) algorithm (Hughes, Ferencz, and Hallquyist 1987) can each be specified by an Operation. An Operation can specify either a symbolic derivation or a numeric computation. For an operation, the variable on the left-hand side is its *output data object*, while those on the right-hand side are its *input data object*. A PIER *Dataobject* specifies numerical values associated with elements/nodes organized in a structured fashion (e.g. matrix, vector).

- **Module**: A Module consists of a sequence of Operations with no entry or exit points except at the beginning and at the end of the module. In other words, control flow enters at the beginning and leaves at the end of a Module. Fig. 3 illustrates Module specifications. In the example, `Module, In, Out, Begin, End` are keywords whereas `VecInnerVec` and `MatTimesVec` represent Operations (for numerical computations).

```
Module CSSA, (In:(a,r,z,p),Out:(Temp1,Temp2))
   Begin
     Temp1 = VecInnerVec(r,z)
     u  = MatTimesVec(a,p)
     Temp2 = VecInnerVec(p,u)
   End
```

Figure 3: PIER Module Specification Example

- **Algorithm**: An Algorithm is specified by combining Modules with f77 constructs. PIER also supplies, from its knowledge-base, certain standard algorithms that can be used directly. Fig. 4 illustrates an example of Algorithm specifications.

```
PIER Algorithm Specification Example
Algorithm:Foo, (In:(a,b),Out:(x))
     Begin
     . . . f77 code . . .
           .
C       Module Call.
    <<(Temp1,Temp2)=Module(CSSA,a,r,z,p)>>
           .
     . . . f77 code . . .
```

Figure 4: PIER Algorithm Specification Example

- **FEA Solution Step**: The breakup of the FEA solution process recognizes eight solution steps. Each step can be solved by more than one (symbolic/numerical) procedure and expects a fixed set of input quantities and computes a fixed set of results. Depending on the problem formulation, the algorithm used for a solution step may be different. Some standard algorithms such as *Gauss quadrature*, *Gaussian elimination* and *preconditioned conjugate gradient* are built into PIER. Others can be supplied by the user through PIER input specifications.

To derive/generate desired FEA computations (symbolic formulae/f77 code) the user must first specify the element mesh, the element properties, the data arrays for material matrix and nodal coordinates. This is followed by the specifications to derive desired element formulae. We now give an example (Fig. 5) where the problem domain is divided into 256 linear triangular elements. Total number of nodes in the mesh is 153. The local degrees of freedom at each node is 1. For complete syntax and detailed examples for PIER input specifications the reader is referred to (Sharma and Wang 1991b).

```
C Defining Triangular Element Mesh.
m = Mesh(Dim:2,Nodes:153,Elements:256)
e = Element(Ldim:1,Nodes:3,Shape:Triangle)
Dataobject x,Name:XNodalCoordinate
Dataobject y,Name:YNodalCoordinate
Dataobject enm,Name:ElementNodalMatrix
Dataobject m,Name:MatMax

C Deriving element approximations.
h=DeriveShape(Algorithm:Polynomial,e)
b=DeriveBMatrix(Algorithm:Displacement,d,h)

C Generating Numerical Code for a FEA Step.
(x)=SolveSystem(Algorithm:Pcg,k,r,File:foo)
```

Figure 5: PIER Input Specification Example

Synthesis Process

In PIER the FEA programs are synthesized by the method of *composition of program components*. The Operations (in PIER knowledge-base), Modules (User-defined) and Algorithms (User-defined) represent program components in the increasing order of hierarchy. The PIER code generator incrementally refines input-specifications into FEA programs. The code generation is overviewed in Fig. 7 and consists of following phases

1. **Parsing Input Specifications**
2. **Problem Definition**
3. **Code Generation**

In the first phase various input specification constructs are identified and translated into PIER internal Common Lisp function calls. The Algorithm template is recognized and preserved. The functions related to the problem definition (parameters of FEA mesh/element and symbolic derivation of element properties) are executed first. This assigns appropriate values to control variables in the environment. The first phase also identifies and analyses PIER Modules, Module Calls and input/output data object specifications. The user-specified element approximations are derived using symbolic mathematical computations (using AKCL-MAXIMA) and the MAXIMA internal representations are translated into equivalent Common Lisp expressions.

The code generation phase generates code for each Module and constituting Operations. Each Module Call in the Algorithm specification generates code for a Module. Symbolic expressions, if any, appearing in the Module body are translated into equivalent Operation specifications. The code for a Module is generated as a set of f77 subroutine calls and the corresponding subroutines. After generating code for all of the Modules referred to by Module Calls, the Gencray (Weerawarana and Wang 1989) translator is called to translate the generated Common Lisp forms into equivalent f77 statements and the holes in the Algorithm template are filled appropriately. In the following subsections we describe code generation from Module and Operation followed by an overview of problem solving with PIER.

Module Code Generation

The code generation from PIER Modules is modeled by *flowgraphs*. A flowgraph is a collection of *flownodes*, which represent task instances and directed *edges*, which represent data dependencies among flownodes. The code generator *derives the flowgraph* representation from the sequence of Operations specified in the Module body and *schedules* the flowgraph onto the target architecture. Operations and data objects form flownodes and edges of the flowgraph respectively. The flownodes, thus, represent coarse grained tasks which have unique cost-models and may be assigned different execution styles.

The schedular of the code generator takes as input a flowgraph, a processor count, and Module execution style (optional). The schedular assigns appropriate number of processors and an execution style to each Operation. The execution style must be consistent with the input and output data objects of the Operation. The schedule should conform to the cost-models of Operations and respect the partial order represented by the flowgraph. The overall objective is to produce a schedule with the lowest total cost.

Details of the parallel code generation can be found in (Sharma and Wang 1990) and (Sharma 1992).

Operation Code Generation

Many of PIER Operations involve regular computations and are internally parallelized in the data parallel fashion. Execution styles (*BlockParallel, Assembled* etc.) refers to methods of partitioning the input/output data objects. The user-specified quantities and output of the schedular are used to refine the algorithm schemas, which implements the Operations. PIER accepts input data objects organized in various specialized storage strategies (e.g. Symmetric Matrix, Banded Matrix etc.). The appropriate data reference mapping are automatically generated in the output code.

Problem Solving with PIER

To use PIER in practice, the first step is to prepare a mathematical model describing the physical situation. The modeler, then, prepares the weak statement followed by dividing the problem domain in a series of elements. Here, we are not concerned with the discretization process and assume that one of the several available domain decomposition software tools is used. However, domain discretization data (i.e. element type, nodal coordinates, list of nodes associated with each element) are to be organized in an appropriate fashion for PIER consumption. To derive computations for any FEA solution steps, the modeler must first define the element mesh. This is followed by input specifications for FEA solution steps which include desired quantities/methods for symbolic derivation and numerical computation. If the desired numerical algorithm is not part of the PIER's knowledge-base, the complete algorithm has to be expressed in input specifications. The modelers can use PIER to generate f77 code for FEA solution steps. The generated code, if desired, can be executed in conjunction with an existing FEA package. The process is outlined in Fig. 6.

Issues

As indicated earlier, in the present work we are focusing on two issues, that we consider critical, in FEA code generation: programmable code generator and code generation for multiple parallel architectures.

Progammable Code Generator

FEA solution method involves symbolic mathematical manipulation and numerical computation with large data sets. To solve a FEA solution step the modelers make choices for the domain mesh, element approximations, and numerical solution algorithm. The choices made are based on: the characteristics of the posed model, target (parallel) computer, and numerical convergence properties. Therefore the FEA solution programs are highly specialized. Code generation systems which would cover all possible cases are bound

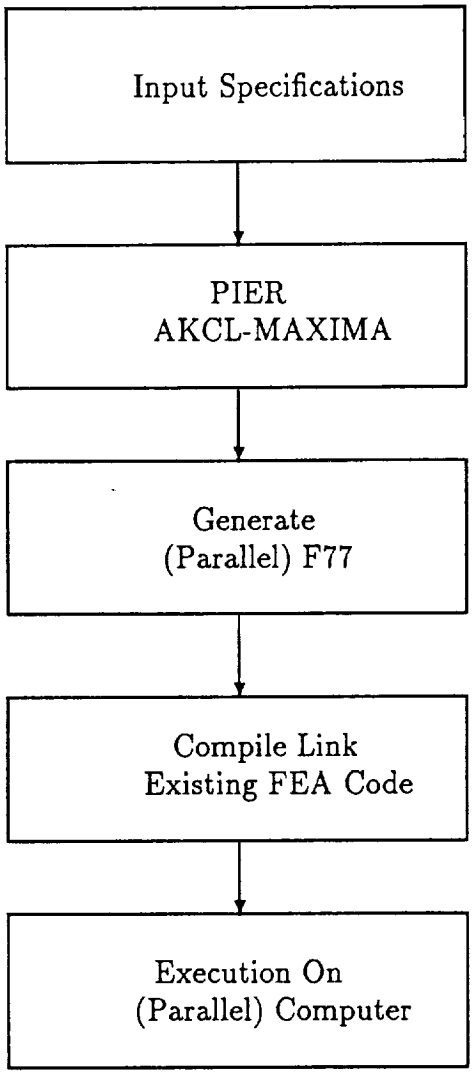


Figure 6: Problem Solving with PIER

to be large, difficult to maintain, and slow. Our approach is to identify and create library of Operations (and possibly Modules), which can generate program components for specific situations. These components are reusable among FEA solution procedures. A solution algorithm can be fabricated using library components and non-FEA specifications in standard f77. The users can customize the generators to their specific needs.

Parallel Code Generation

A major open issue in parallel code generation is the modeling of architecture and the representation of machine specific parallel programming knowledge. In PIER, the computations are represented in an architecture independent formulation (flowgraphs). The Operations generate instances of flowgraphs and attach appropriate code segments to the flownodes. The flowgraph schedular is machine-specific and is the back-end of PIER. Parallel programming rules are transformations from flowgraphs representations to equivalent f77 templates.

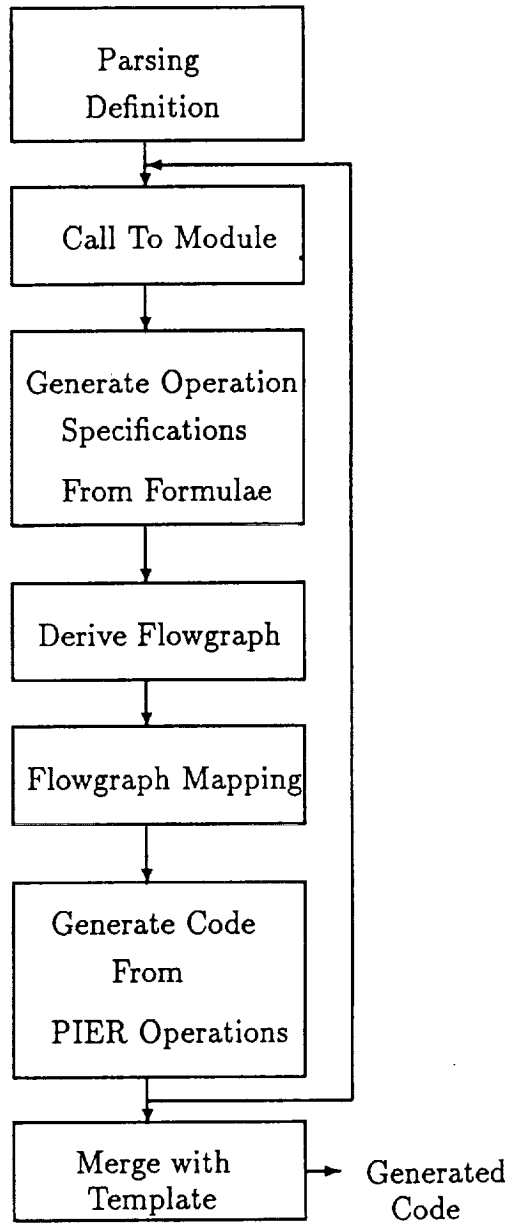


Figure 7: Code Generation Scheme

References

Chang, T. Y. 1986, NFAP - A Nonlinear Finite Element Program, Vol. 2 - Technical Report, College of Engineering, University of Akron, Akron, OH.

Fritzson, Peter and Fritzson Dag, 1991, The Need for High-Level Programming Support in Scientific Computing Applied to Mechanical Analysis, Research Report LiTH-IDA-R-91-04, Department of Computer and Information Science, Linkoping University, S-158 83, Linkoping, Sweden.

Sharma, N. 1988b, Generating Finite Element Programs for Warp Machine, Proceedings of ASME Winter Annual Meeting, Chicago, IL., Nov. 25-28.

Sharma, N., and Wang, Paul S., 1990, Generating Parallel Finite Element Programs for Shared-Memory Multiprocessors, Symbolic Computation and Their Impact on Mechanics, PCP-Vol. 205, *A. K. Noor, I. Elishakoff and G. Hulbert*, Editors, The American Society of Mechanical Engineers, New York.

Sharma, N. and Wang Paul S., 1988a, Symbolic Derivation and Automatic Generation of Parallel Routines for Finite Element Analysis, Lecture Notes in Computer Science, Gianni, P. (Ed.), Proceedings International Symposium on Symbolic and Algebraic Computations 33–56, Rome, Italy.

Sharma N. 1991a, Generating Finite Element Programs for Multiprocessors, Fifth SIAM Conference on Parallel Processing for Scientific Computing, Houston, TX.

Sharma, N. and Wang, P. S. , 1991b, High-level User Input Specifications for Finite Element Code Generation, Conference on Design and Implementation of Symbolic Computation Systems (DISCO), April 13–15, 1992, University of Bath, Bath, UK.

Sharma, N. 1992, The PIER Parallel FEA Program Generator, In Preparation.

Weerawarana, Sanjiva and Wang, Paul S., 1989, Gencray: User's Manual, Department of Mathematics and Computer Science, Kent State University, Kent.

Wang, P. S. 1986, FINGER: A Symbolic System for Automatic Generation of Numerical Programs for Finite Element Analysis, *Journal of Symbolic Computation*, Vol. 2, pp. 305-316.

Steinberg, S. and Roache, P. J., 1990, Using MACSYMA to write finite-volume based PDE Solvers, Symbolic Computation and Their Impact on Mechanics, PCP-Vol. 205, *A. K. Noor, I. Elishakoff and G. Hulbert*, Editors, The American Society of Mechanical Engineers, New York.

Allen, J. R. and Kennedy, K., 1985, PFC: a program to convert Fortran to parallel form, *Supercomputers: Design and Applications*, K. Hwang, editor, IEEE Computer Society Press, pp 186-205.

ParaScope Editor.

Kuck, D. J. 1978, *The Structure of Computers and Computations*, Volume 1, John Wiley and Sons, New York.

Russo, Mark F., Peskin, Richard L. and Kowalaski, A. Daniel, 1987, Using Symbolic Computation for Automatic Development of Numerical Programs. *Coupling Symbolic and Numerical Computing in Expert Systems, II.*

Rice, John R., and Boisvert, Ronald F., 1985, *Solving Elliptical Problems Using ELLPACK*, Springer Series in Computational Mathematics 2, Springer-Verlag, New York.

Kant, E., Daube, F., MacGregor, W., and Wald, J., 1990, Synthesis of Mathematical Modeling Programs. Technical Report, TR–90–6, Schlumberger Laboratory for Computer Science, Austin, TX 78720.

Cook, Grant O. 1990, ALPAL, a Program to Generate Simulation Codes from Natural Descriptions. Technical Report UCRL–102076, Lawrence Livermore National Laboratory, L-35, Livermore, CA 94551.

Hirayama, H., Ikeda, M., and Sagawa, N., 1991, Solution Functions of PDEQSOL (Partial Differential EQuation SOlver Language) for Fluid Problems, In Proceedings of Supercomputing, pages 218-227. ACM Press, November 1991.

Zienkiewicz, O. C. 1980, *The Finite Element Method in Engineering Science*, Mc-Graw Hill, London, pp. 129-153.

Hughes, T. J. R., Ferencz, R. M. and Hallquyist, J.O., 1987, Large-scale Vectorized Implicit Calculations in Solid Mechanics on Cray X-MP/48 Utilizing EBE Preconditioned Conjugate Gradients, *Computer Methods in Applied Mechanics and Engineering*, Vol. 61, No. 2, 1987, pp. 215-248.

Winget, J. M. and Hughes, T. J. R., 1985, Solution Algorithms for Nonlinear Transient Heat Conduction Analysis Employing Element-by-Element Iterative Strategies, *Computer Methods in Applied Mechanics and Engineering*, Vol. 52, pp. 711-815.

KNOWLEDGE MODELING FOR SOFTWARE DESIGN

Mildred L G Shaw & Brian R Gaines

Knowledge Science Institute
University of Calgary
Calgary, Alberta, Canada T2N 1N4
mildred@cpsc.ucalgary.ca, gaines@cpsc.ucalgary.ca

Abstract: This paper develops a modeling framework for systems engineering that encompasses systems modeling, task modeling, and knowledge modeling, and allows knowledge engineering and software engineering to be seen as part of a unified developmental process. This framework is used to evaluate what novel contributions the 'knowledge engineering' paradigm has made, and how these impact software engineering.

INTRODUCTION

In the knowledge acquisition community the development of tools for eliciting knowledge from experts has come to be seen as a 'knowledge modeling' exercise in which human practical knowledge is modeled within the computer (Gaines, Shaw and Woodward, 1992). It has been suggested that a common factor underlying all knowledge-based systems, including software design systems, is that they contain qualitative world models, and that we can gain insights into the structure of knowledge bases and knowledge engineering by classifying the types of models involved (Clancey, 1989). These considerations suggest that a classification of the sources and types of models developed in system engineering may be used to provide a framework within which knowledge engineering and software engineering methodologies and tools can be analyzed and compared.

One might view the replication of human expertise in a knowledge-based system as involving the elicitation of the *mental models* of the human experts involved (Gentner and Stevens, 1983). However, we do not have direct access to these models, and must create *conceptual models* of them through communication with the expert (Norman, 1983). The representations made by the knowledge engineer are not isomorphic to structures in the mind of the expert (Compton and Jansen, 1990). Within this framework, one can view knowledge engineers, or automated knowledge acquisition systems interacting with the expert, as accessing and developing the expert's conceptual models. Some parts of these models may be pre-existent, particularly if the expert has a teaching role, but other parts will come into being as a result of the knowledge acquisition process.

The distinction that Norman introduces between mental models and conceptual models, and the dubious status of mental models in themselves, suggests that a useful framework for the analysis of knowledge engineering may be developed through the analysis of the sources and types of conceptual model available to the knowledge engineer rather than focusing only on the mental processes underlying expertise. The situation of the introspective expert who can communicate his or her 'knowledge' well, may be treated as one where the 'knowledge engineering' and 'expert' roles are operating effectively together within the same person. The situation of the expert from whom knowledge is being 'elicited' actually building a new model on the basis of his or her skills through the process of elicitation may be treated as one where the conceptual model is developed as part of the process of knowledge engineering. In adopting the conceptual modeling perspective we do not exclude previous viewpoints, but rather supplement them with complementary perspectives.

A MODELING FRAMEWORK FOR INFORMATION SYSTEM DEVELOPMENT

It is customary in expert system development, to assume that the expert has already constructed such models or may be in a privileged position to do so through self-observation and introspection, and these may be elicited by direct communication between knowledge engineer and expert. Additionally, the knowledge engineer may derive models from other experts, from the literature, and from the application of principles allowing performance skills to be derived from deep knowledge. The final knowledge-based system development involves the synthesis of these many models and the encoding of them to become an operational knowledge-based systems emulating the desired expertise.

Thus, the knowledge engineer, or knowledge engineering team and tools, has access to multiple sources of data through various channels and uses these to develop a variety of conceptual models. Figure 1 shows the major conceptual models that may be developed in knowledge engineering, distinguished by their sources, and indicating some of the knowledge engineering processes and skills involved. This figure attempts to be comprehensive, showing knowledge sources not only in association with the expert and his or her behavior, but also knowledge derived from others, the literature and through the application of laws and principles.

Figure 1 is an accurate representation of what is typically involved in knowledge engineering for a knowledge based system development nowadays. It uses any source of knowledge that is available for system development, not just the practical reasoning of the expert, and hence exemplifies the "second type" of knowledge engineering cited above (Feigenbaum, McCorduck and Nii, 1988). However, it still has a major, and irreducible component of the first type representing the central expert systems paradigm. What is significant is the way in which the two approaches are synthesized, and also the way in which many components of the "second type" of activity are already part of modern systems and software engineering. This is the basis of a much wider synthesis than that between two forms of knowledge engineering.

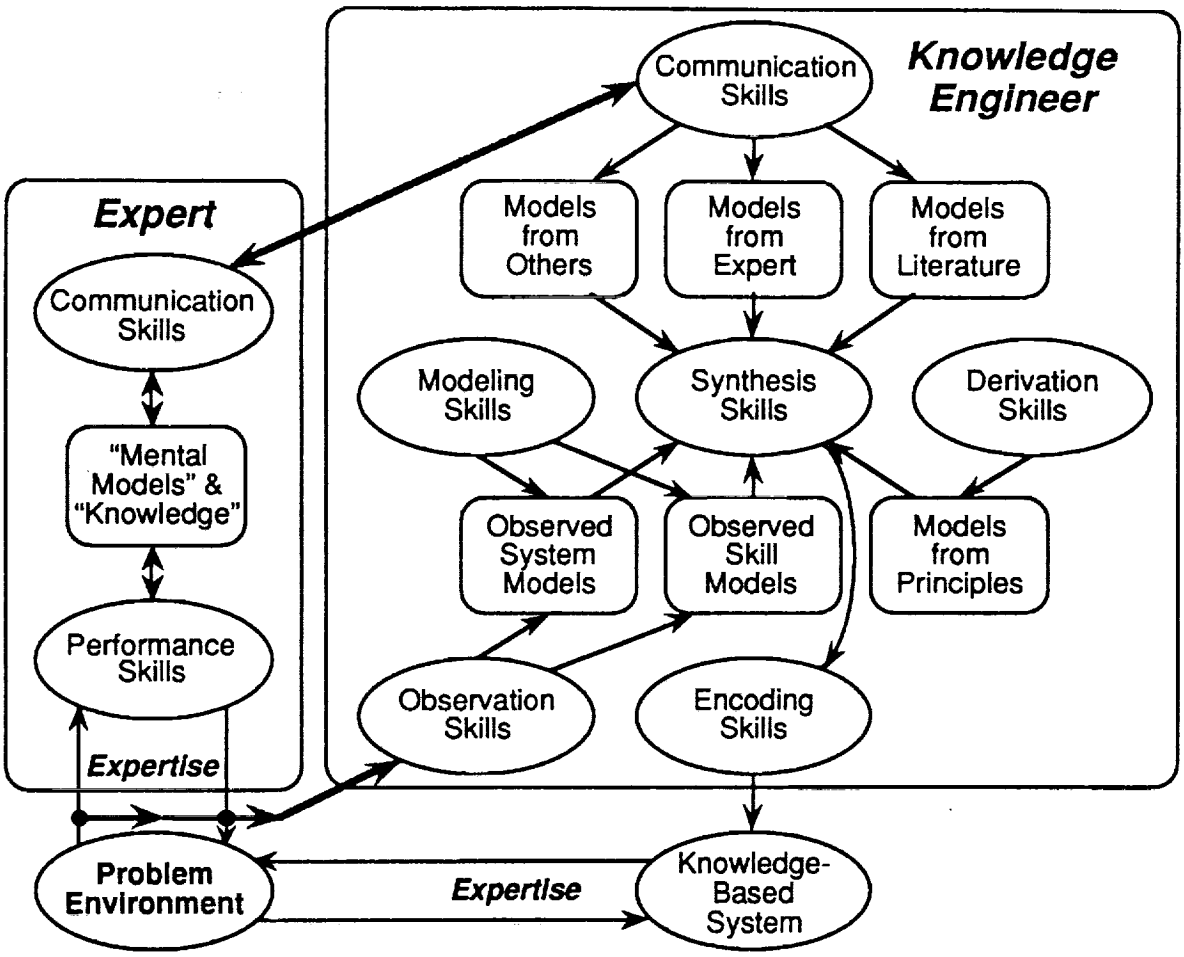


Fig.1 Modeling processes in knowledge engineering

A MODELING FRAMEWORK FOR INFORMATION SYSTEM DEVELOPMENT

The discussion of the preceding sections and the range of modeling processes shown in Figure 1 provide an overall framework for systems engineering in terms of the sources and types of models involved. Within such a framework it should become only a matter of internal classification and terminology that a method is part of a 'knowledge engineering' or a 'software engineering' approach, rather than a resultant system classification.

Figure 2 presents a modeling framework for knowledge acquisition methodologies, techniques and tools based on the distinctions already discussed and the incorporation of system analysis and software engineering procedures. In the leftmost column are the knowledge sources in terms of systems and modeling schema already discussed with the addition, at the top, of 'objective models' as a term for the formally specified operational models. In the column to the right of this are the processes giving access to these models. These processes are shown as mediating between the systems and models involved, deriving from and generating, the hierarchical relation between the systems and models in the leftmost column.

In the next column on the right are shown the knowledge acquisition procedures appropriate to each of the access processes. These generate data and knowledge bases as shown to their right, which are in one-to-one correspondence with the original systems and models in the leftmost column. In the rightmost column are shown analysis and synthesis techniques that draw on these databases to generate the computational knowledge base, and also mediate between them generating one form of data or knowledge from another. These combine with synthesis techniques that integrate the results of analysis and of derivations from various knowledge sources to synthesize a computational knowledge base.

Thus the overall schema consists of five types of component:

1. *Systems and modeling schema*: the problem environment, performance skill to be emulated, expert's mental models, knowledge engineer's conceptual models, and, possibly, objective models.
2. *Access processes:* instrumentation of the target system, the expert's interaction with it, his or her introspection about the skill, communication about it, and its expression in formal terms as objective knowledge.

3. *Knowledge acquisition procedures*: observation of the target system, observation of the expert's behavior, elicitation procedures, discourse procedures, formalization procedures, and implementation procedures.
4. *Data and knowledge bases:* database of system data; database of behavioral data; informal knowledge base; formal knowledge base; computational knowledge base; objective models.
5. *Analysis and synthesis procedures*: classical system identification can be used to build system models from observation data; empirical induction and case-based clustering can be used to build skill models from behavioral data; conceptual organization and linguistic analysis techniques can be used to build a formal, or structured, knowledge base from an informal, or intermediate, one; knowledge modeling techniques can be used to represent the formal knowledge base in computational form; and logical deduction from laws and principles may be used to provide some knowledge about a system and this, together with the results of data analyses from various sources needs to be integrated to form a computational knowledge base.

All the earlier stages of analysis are shown as normally creating data at the next level but also as potentially creating computational systems in their own right.

Figure 2 illustrates the way in which knowledge engineering as a system design methodology is sandwiched between two classical approaches to system engineering. At the bottom is the path to system design through instrumentation, data collection and system identification. At the top is the path to system design through existing objective knowledge of the physical world allowing explication of particular requirements to lead directly to implementation. The middle layers represent the enrichment of the design process when we draw on human skills as exemplars of the system to be designed. Such a process has been common informally in engineering design, and knowledge engineering may be seen as formalizing it now that computer technology makes it feasible to develop knowledge-based systems operationalizing human expertise.

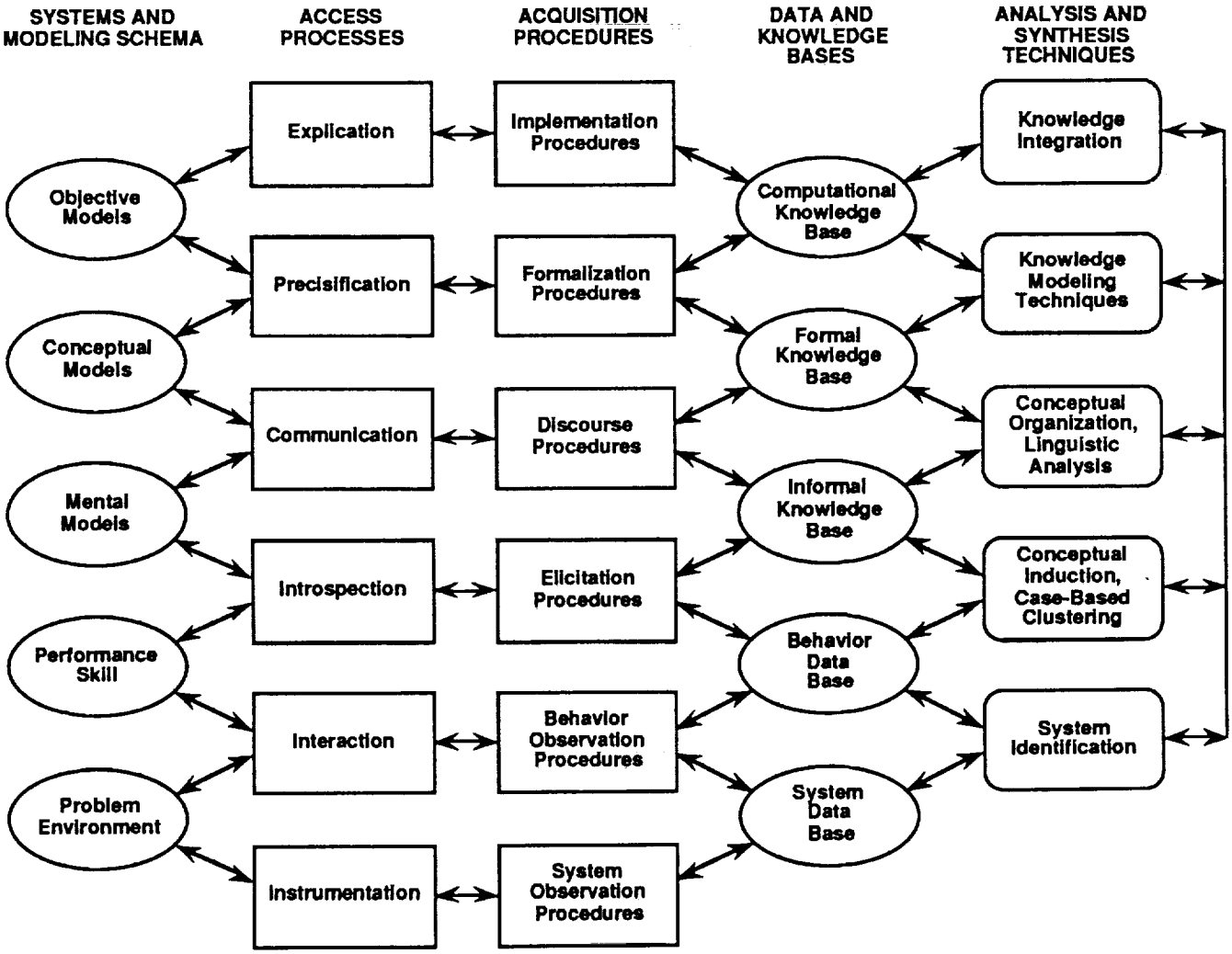


Fig.2 A hierarchical framework for knowledge acquisition

KNOWLEDGE ACQUISITION ISSUES IN TERMS OF THE FRAMEWORK

It is clear that a catchall term such as 'interviewing' does not designate a monolithic technique in terms of the framework of Figure 2. When we interview an expert we may be operating at any level of the hierarchy and may be supporting any one of the many processes shown. All that we can say about interviewing in general is that a flow of linguistic information is involved—it is the content of that flow that determines the type of knowledge engineering involved. The expert may provide observations of the system, observations of his or her own problem solving behavior, introspection about aspects of his or her mental models, statements about his or her conceptual models of any aspect of the situation, and statements of formal or even computational models relating to the situation.

Specific knowledge acquisition techniques are characterized by their vertical and horizontal locations within the framework. For example, protocol analysis involves data collection for the behavior data base through observation of interaction at one level or elicitation of introspection at the next. The behavior database is then subject to statistical system identification or to conceptual induction and clustering. The data collection methodology in protocol analysis may easily slip into the elicitation of not just a protocol but also an explanatory commentary which belongs in the informal knowledge base and is subject to linguistic analysis. Thus, applications of protocol analysis may involve multiple levels and activities that are confusing unless seen as organized within the framework.

Analytical tools such as induction and clustering algorithms have a well-defined location in the framework as analysis techniques providing a model creation technology. Their differentiation comes from what level, or levels, they can accept data, and at what level, or levels, they create data. A major focus in machine learning research for several years has been to create models at the knowledge level, conceptual structures rather than rules. To the extent that all the analytic techniques involved do this, the problem becomes one of integration of conceptual structures. However, it is more usual to find that the analytic tools create data or knowledge at different levels and further processing is required before integration is possible.

Methodologies such as KADS (Akkermans, Harmelen, Shreiber and Wielinga, 1992) that provide a structured software engineering approach to knowledge engineering are focused at the penultimate level of applying formalization procedures to derive a formal knowledge base through making conceptual models precise. KADS focuses on the detailed structure of a formal problem solving architecture within which to operationalize the results of knowledge acquisition rather than on the processes of knowledge acquisition themselves. It may be seen as providing a formally specified 'virtual machine' well-suited to the range of system developments that have come to be classified under the heading of 'knowledge-based systems.' Less formally, one can say that it provides a 'high-level language' in contrast to the 'machine languages' provided by expert system shells.

Knowledge acquisition methodologies such as those stemming from personal construct psychology (Shaw, 1980) that are based on a cognitive model of intelligent agents are focused on the middle levels in Figure 2, modeling the way in which mental models mediate between conceptual models and performance skills. Clearly any well-founded cognitive psychology has a potential role to play in knowledge acquisition that is strictly within the 'expert systems' paradigm of modeling the expert rather than the system. However, to be useful the psychology must result in operational models on the one hand and support methodologies giving access to its hidden variables on the other. Personal construct psychology has been particularly attractive in these respects because, even though it is a constructivist model, it takes a positivist, axiomatic approach based on a few well-defined primitives that correspond to a formal intensional logic (Gaines and Shaw, 1992), and is well-supported by practical tools (Boose and Bradshaw, 1987; Shaw and Gaines, 1987; Shaw and Gaines, 1989).

The interface between cognition and formalization for people is mediated through language and knowledge acquisition support is required for the communication and discourse procedures and analysis level in Figure 2. Current knowledge acquisition tools addressing this level range from those focusing on the inter-translation of restricted natural language and knowledge representation frames such as SNOWY (Gomez and Segami, 1990), to those providing support for human classification of natural language components in terms of knowledge level primitives such as Cognosys (Woodward, 1990). Improved natural language processing must have a very high priority in the support of the complete range of knowledge acquisition processes in the framework of Figure 2.

Classical system analysis focuses on the collection and analysis of system and behavior data at the lower levels of Figure 2. In complex system development the other levels play their part, but the basic assumption has been that the final system design is grounded in accurate models of the environment in which the system is to operate and in precise 'requirements specifications' corresponding to the top level goals of the human agents involved. The implementation is quite separate from the system analysis and design because conventional programming languages do not provide knowledge-level constructs supporting human understanding of their operation.

In this respect, the framework of Figure 2 may be seen as an extension to classical system analysis appropriate to knowledge-based systems where very high level languages at the 'knowledge level' are being used for the implementation to provide this support of human understanding.

CONCLUSIONS

A complete account of system engineering acquisition for modern advanced information systems requires the integration of classical system analysis, cognitive modeling of intelligent agents, linguistic analysis of text and discourse, and a rich formal language at the knowledge level. This integration would provide us with a system development methodology adequate to cope with the increased expectations of those specifying requirements for knowledge-based systems.

However, note that the knowledge level language alone is only a target for specification. On the one hand it needs to be made operational as computational knowledge. On the other it needs to maintain an effective ongoing relation with the knowledge processes that drive it, many of which are those of active human agents forming an essential component of the ongoing system operation. Knowledge acquisition should not be seen as part of the system design process only. Knowledge is dynamic and changing, and acquisition, maintenance and upgrading must merge into one process that is fully supported as an ongoing system operation. In particular, the cognitive aspects of much of the knowledge must continue to be recognized and supported in the ongoing system operation. Formalization cannot be at the expense of human understanding. On the contrary, effective formalization should lead to enhanced human understanding. This is the greatest challenge in the development of an effective knowledge-based systems technology. The objective is not just emulation of isolated human peak performance, but rather the emulation of the total human ability to develop, adapt and maintain that performance in a dynamic and uncertain environment.

ACKNOWLEDGEMENTS

This work was funded in part by the Natural Sciences and Engineering Research Council of Canada. We are grateful to our colleague Brian Woodward for discussions that have influenced this paper.

REFERENCES

Akkermans, H., Harmelen, F. van, Shreiber, G. and Wielinga, B. (1992). A formalisation of knowledge-level models for knowledge acquisition. **International Journal of Intelligent Systems** to appear.

Boose, J.H. and Bradshaw, J.M. (1987). Expertise transfer and complex problems: using AQUINAS as a knowledge acquisition workbench for knowledge-based systems. **International Journal of Man-Machine Studies** 26 3-28.

Clancey, W.J. (1989). Viewing knowledge bases as qualitative models. **IEEE Expert** 4(2) 9-23.

Compton, P. and Jansen, R. (1990). A philosophical basis for knowledge acquisition. **Knowledge Acquisition** 2(3) 241-258.

Feigenbaum, E., McCorduck, P. and Nii, H.P. (1988). **The Rise of the Expert Company**. New York, Times Books.

Gaines, B.R. and Shaw, M.L.G. (1992). Basing knowledge acquisition tools in personal construct psychology. **Knowledge Engineering Review** to appear.

Gaines, B.R., Shaw, M.L.G. and Woodward, J.B. (1992). Modeling as a framework for knowledge acquisition methodologies and tools. **International Journal of Intelligent Systems** to appear.

Gentner, D. and Stevens, A., Ed. (1983). **Mental Models**. Hillsdale, New Jersey, Erlbaum.

Gomez, F. and Segami, C. (1990). Knowledge acquisition from natural language for expert systems based on classification problem-solving methods. **Knowledge Acquisition** 2(2) 107-128.

Norman, D.A. (1983). Some observations on mental models. **Mental Models**. Hillsdale, New Jersey, Erlbaum.

Shaw, M.L.G. (1980). **On Becoming A Personal Scientist: Interactive Computer Elicitation of Personal Models Of The World**. London, Academic Press.

Shaw, M.L.G. and Gaines, B.R. (1987). KITTEN: Knowledge initiation & transfer tools for experts and novices. **International Journal of Man-Machine Studies** 27(3) 251-280.

Shaw, M.L.G. and Gaines, B.R. (1989). A methodology for recognizing conflict, correspondence, consensus and contrast in a knowledge acquisition system. **Knowledge Acquisition** 1(4) 341-363.

Woodward, B. (1990). Knowledge engineering at the front-end: defining the domain. **Knowledge Acquisition** 2(1) 73-94.

The Use of Typed Lambda Calculus for Comprehension and Construction of Simulation Models in the Domain of Ecology

Michael Uschold, Ph.D.
University of Edinburgh

1990

Abstract

We are concerned with two important issues in simulation modelling: *model comprehension* and *model construction*. Model comprehension is limited because many important choices taken during the modelling process are not documented. This makes it difficult for models to be modified or used by others. A key factor hindering model construction is the vast modelling search space which must be navigated. This is exacerbated by the fact that many modellers are unfamiliar with the terms and concepts catered for by current tools.

The root of both problems is the lack of facilities for representing or reasoning about domain concepts in current simulation technology. The basis for our achievements in both of these areas is the development of a language with two distinct levels; one for representing domain information, and the other for representing the simulation model. Equally importantly, we make formal connections between these two levels. The domain we are concerned with is ecological modelling.

This language, called *Elklogic*, is based on the typed lambda calculus. Important features include a rich type structure, the use of various higher order functions, and semantics. This enables complex expressions to be constructed from relatively few primitives. The meaning of each expression can be determined in terms of the domain, the simulation model, or the relationship between the two. We describe a novel representation for sets and substructure, and a variety of other general concepts that are especially useful in the ecological domain. We use the type structure in a novel way: for controlling the modelling search space, rather than a proof search space.

We facilitate model comprehension by representing modelling decisions that are embodied in the simulation model. We represent the simulation model separately from, but in terms of a domain model. The explicit links between the two models constitute the modelling decisions. The semantics of Elklogic enables English text to be generated to explain the simulation model in domain terms.

Inherent in this is a new approach to model construction which we have implemented in a computer program called ELK. Users build up a sequence of models, each being used to identify and constrain the important modelling decisions for the next one. The first model consists of general domain information (*e.g.* forestry). The second is a description of the particular situation of interest (*e.g.* some forest). Finally a simulation model of that situation is constructed. This approach enables users to communicate in familiar terms as well as significantly reducing the number of decisions that have to be made at any point. Constructing simulation models this way enables them to be self-documenting; this facilitates model comprehension.

omit

Relevant Publications

Mike Uschold
March 17, 1992

[3] gives full details of motivation, theory and implementation of ELK.

[1] describes an early version of Elklogic.

[4] describes the overall methodology used, but skims over the details of the logic language.

[5] is a revised and extended version of [4]. The former is described using more domain independent terminology, and is tailored for the model management community in the field of decision support systems.

[6] gives a broad overview of the overall project (ECO) in which this work took place.

[2] gives full details of the ECO project (excluding ELK).

References

[1] A. Bundy and M. Uschold. The use of typed lambda calculus for requirements capture in the domain of ecological modelling. Research Paper 446, Dept. of Artificial Intelligence, Edinburgh, 1989. Submitted to Logic and Computation.

[2] D. Robertson, A. Bundy, R. Muetzelfeldt, M. Uschold, and M. Haggith. *Eco-Logic: Logic-based approaches to Ecological Modelling.* MIT Press, 1991.

[3] M. Uschold. *The Use of Typed Lambda Calculus for Comprehension and Construction of Simulation Models in the Domain of Ecology.* PhD thesis, Dept. of Artificial Intelligence, Edinburgh, 1990.

[4] M. Uschold. The use of domain information for comprehension and construction of simulation models. Research Paper 534, Dept. of Artificial Intelligence, Edinburgh, 1991.

[5] M. Uschold. The use of domain information and higher order logic for model management. In Holsapple and Whinston, editors, *Recent Developments in Decision Support Systems*, page ?? Springer-Verlag, 1992. forthcoming.

[6] M. Uschold, D. Robertson, A. Bundy, and R. Muetzelfeldt. Helping inexperienced users to construct simulation programs: An overview of the ECO project. In S. Vadera, editor, *Expert System Applications*, pages 117–132. Sigma Press, 1989. This is a revised and extended version of a paper published in 'Research and Development in Expert Systems 4', BCSES 1987; Also available as D.A.I. Research paper 338.

Knowledge-Based Design of Generate-and-Patch Problem Solvers that Solve Global Resource Assignment Problems

Kerstin Voigt*
Computer Science Department, Rutgers University
New Brunswick, NJ 08903
voigt@cs.rutgers.edu

Abstract

We present MENDER, a knowledge based system that implements software design techniques that are specialized to automatically compile generate-and-patch problem solvers that satisfy global resource assignments problems. We provide empirical evidence of the superior performance of generate-and patch over generate-and-test, even with constrained generation, for a global constraint in the domain of "2D-floorplanning". For a second constraint in "2D-floorplanning" we show that even when it is possible to incorporate the constraint into a constrained generator, a generate-and-patch problem solver may satisfy the constraint more rapidly. We also briefly summarize how an extended version of our system applies to a constraint in the domain of "multiprocessor scheduling".

Introduction.

The MENDER project presented here is aimed at developing techniques to automatically compile generate-and-patch problem solvers; these problem solvers efficiently construct solutions that satisfy a conjunction of interacting constraints. MENDER is a part of the larger KBSDE effort towards automating knowledge-based design of algorithms [Tong, 1991]. MENDER builds on recent successes in automatically compiling conjunctions of constraints on composite structures into *"constrained" generate-and-test* problem solvers [Braudaway, 1991]. The constrained generator is the result of incorporating into the generator constraints that constrain local parts of the composite structure ("local constraints"). The conjunction of constraints may feature other constraints which restrict all or most parts of the artifact simultaneously ("global constraints"), a property that prevents their successful incorporation. These constraints could be satisfied by placing testers after the constrained generator. However, the resulting generate-and-test problem solver may still perform grossly inefficient.

The MENDER research capitalizes on the observation that once a complete composite artifact has been generated, frequently a small number of local, well-directed modifications ("patches") to the artifact suffice to produce a solution to a constraint. We have developed and implemented the MENDER compiler which automatically builds such *generate-and-patch* problem solvers from an inefficient generate-and-test problem solver. MENDER automatically compiles a *hillclimbing* search component, called a *"patcher"*, and interfaces it with the generator. Hillclimbing patchers are defined by the set of *patching operators* (i.e. parameter value changes) and an *evaluation function* that measures progress towards constraint satisfaction. We show that cost-effective generate-and-patch problem solvers are possible with patchers that have the following two properties. (1) Patching operators do not result in violations of constraints that have been satisfied through generation. (2) Patching searches the space of parameter value changes in a *constraint-oriented* fashion; i.e. when choosing the next move, changes that promise higher degrees of constraint satisfaction are prefered over those that yield no improvement, or worsen the state.

The MENDER research has focused on the automatic compilation of *hillclimbing patchers* that are specialized to satify global constraints that in some abstract sense involve the assignment of "resources" to "consumers". *Resource assignment problems* (RAPs) are constraints that constrain the nature of assignment between components of a composite "consumer structure" and a composite "resource structure". Our con-

*The research reported here was supported in part by the National Science Foundation (NSF) under Grant Number IRI-9017121, in part by the Defense Advanced Research Projects Agency (DARPA) under DARPA-funded NASA Grant NAG2-645, in part by DARPA under Contract Number N00014-85-K-0116, in part by NSF under Grant Number DMC-8610507, and in part by the Center for Computer Aids to Industrial Productivity (CAIP), Rutgers University, with funds provided by the New Jersey Commission on Science and Technology and by CAIP's industrial members. The opinions expressed in this paper are those of the author and do not reflect any policies, either expressed or implied, of any granting agencies.

vention is that the consumer structure corresponds to the output of generate-and-test or generate-and-patch problem solvers. The resource structure is typically some other *constant* structure that is given as a part of the specification of a RAP. RAPs are of interest to us because they can successfully model significant portions of important and well-known classes of problems (floorplanning, scheduling, n-queens,...). Secondly, they share features that make them conducive to the automatic compilation of efficient generate-and-patch problem solvers.

In this paper, we report on MENDER's success in automatically compiling, and interfacing with a constrained generator, a patcher that satisfies two global constraints in the domain of "2D-floorplanning". The first global constraint which we will refer to as "fill house" constraint is informally defined as

> Constraint "fill house": The rooms in the floorplan must cover the house area completely.

For this constraint we will briefly sketch the steps of MENDER's compilation of a generate-and-patch algorithm, and present the results of a preliminary performance study on the resulting problem solver.

The second constraint – refered to as "no overlap" – is defined as

> Constraint "no overlap": Rooms in the floorplan do not overlap.

We will not sketch the compilation of a generate-and-patch problem solver for "no overlap", but towards the end of this paper we will present performance data on the algorithm that MENDER constructed. The "no overlap" constraint is interesting to us because it lies on the boundary between local and global constraints. The RICK compiler [Braudaway, 1991] has been able to partially incorporate "no overlap" into a generator of floorplans by compiling filters that forward propagate necessary conditions derived from generated rooms to rooms that are to be generated next. We compared the performances of the RICK-compiled constrained generator with the MENDER-compiled generate-and-patch algorithm for "no overlap". We will see that generate-and-patch consistently performed better than constrained generation.

To show generality of MENDER's methods we will also briefly describe how MENDER can automatically compile a generate-and-patch problem solver for a constraint in the domain of "multiprocessor scheduling".

Classification-based compilation of hillclimbing patchers.

In the past, we have already presented a *classification-based* technique to construct hillclimbing patchers [Voigt and Tong, 1989]. The here presented 5-step technique is an improvement over the previous one in that a lattice-like taxomomy of abstract RAP schemas can be exploited in ways that eliminate costly matching and theorem proving. The lattice formed by the 16 abstract RAPs is illustrated in Fig. 1. The least

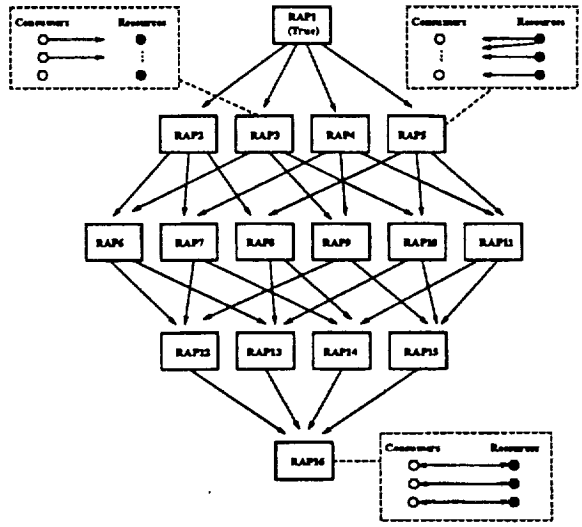


Figure 1: Taxonomy of abstract RAP schemas.

constrained RAP schema, RAP1 (unconstrained, or "True"), is found at the top of the lattice, the most constrained RAP schema, RAP16 (bijection from consumers to resources, and bijection from resources to consumers) forms the bottom of the lattice. RAP2, RAP3, RAP4, and RAP5 are the *basic* RAPs. All RAPs on the remaining layers of the lattice are constructed from the basic ones by conjunction, as indicated by downwards arrows (e.g. RAP6 ≡ RAP2 ∧RAP3). Thus, RAP16 is the conjunction of all four basic RAPs. We will see later how explicit knowledge of these relationships between the RAPs can be exploited in constructing efficient algorithms.

Step 1 – Classifying the constraint. In its first step, MENDER asks the user to define the global constraint, e.g. "fill house", in a guided sequence of questions. The answers to these questions are rephrased as a first-order logic representation of the concrete constraint. In parallel to formulating the concrete constraint MENDER parses a *taxomomy* of abstract RAPs; when all questions have been answered the abstract RAP class for the constraint has also been determined. For example, the "fill house" constraint may be formulated as

$$\forall p \{\text{point-of-rect}(p,Hs) \Rightarrow \exists rm [\text{member}(rm,Fp) \wedge \text{point-of-rect}(p,rm)]\}$$

> "Each point p in the rectangular house (Hs) is assigned to at least one room (rm) point in the floorplan (Fp)."

and is automatically classified as an instance of the abstract resource assignment problem RAP5, which is represented by the abstract first-order logic sentence

below:

$\forall r$ {*substruct(r,Resources)* $\Rightarrow$
$\exists c$ *[substruct(c,Consumers)* $\wedge$ *assign(r,c)]}*

"Each resource r in the resource structure (Resources) is assigned to at least one consumer c in the consumer structure (Consumers)."

MENDER pairs up the terms of the abstract RAP language (e.g. *substruct*, *assign*) with the concrete terms of "fill house" as follows:

p ..... *r*
rm ..... *c*
Hs ..... *Resources*
Fp ..... *Consumers*
point-of-rect(p,Hs) ... *substruct(r,Resources)*
member(rm,FP) ... *substruct(c,Consumers)*
point-of-rect(p,r) ..... *assign(r,c)*

Step 2 – Deriving an evaluation function. Associated with each abstract RAP is a *generic evaluation function* which indicates in abstract terms how to measure degrees of constraint satisfaction by quantifying assignments between consumers and resources. From RAP5 we retrieve the generic evaluation function shown in Fig.2. It computes the number of resources assigned to at least one consumer. MENDER specializes this abstract evaluation function into a concrete evaluation function for "fill house" by instantiating the abstract terms with the corresponding concrete terms. The evaluation function for "fill house" then computes the number of points p in house Hs that are also points in some room rm in floorplan Fp (Fig.2).

Typically MENDER faces a scenario in which a constrained generator exists which already guarantees the satisfaction of several constraints. MENDER's task consists in constructing and interfacing with this generator a patcher which will satisfy an additional global constraint. When some or all of the constraints that have been incorporated into the generator match several of the basic RAP schemas (or conjunctions thereof), then properties of the RAP lattice can be exploited to derive an evaluation function that is more efficient than the one in Fig. 2. For example, imagine that a constrained generator exists that guarantees that all rooms in the floorplan are located *inside* the house (RAP2) and do *not overlap* (RAP4), and and the house is *flat* (RAP3) (i.e. one room point cannot coincide with more than one house point). Then satisfying "fill house" (RAP5) actually implies satisfying the conjunction of these constraints. The conjunction in turn is equivalent to RAP16. We know that RAP16, that is, the conjunction of RAP2, RAP3, RAP4, and RAP5, implies that the number of consumer substructures must be equal to the number of resource substructures. I.e.

$$\text{RAP2} \wedge \text{RAP3} \wedge \text{RAP4} \wedge \text{RAP5} \Rightarrow \text{SIZE}(C) = \text{SIZE}(R),$$

```
Evaluation function for RAP5:

variables: r c out temp;
constants: Consumers Resources
begin-proc
  out <- 0
  forall SUBSTRUCTS r in Resources
    temp <- 0
    forall SUBSTRUCTS c in Consumers
      if ASSIGN(c,r) = true
         then temp <- 1
              exit-forall end-if
    end-forall
    out <- out + temp
   end-forall
   return out
end-proc
```

⇓ *INSTANTIATE* ⇓

```
Evaluation function for ''fill house'':

variables: p rm out temp;
constants: Fp Hs
begin-proc
    out <- 0
    forall POINT-OF-RECT p in Hs
       temp <- 0
       forall MEMBER rm in Fp
          if POINT-OF-RECT(p,rm) = true
             then temp <- 1
                  exit-forall end-if
        end-forall
     out <- out + temp
     end-forall
     return out
end-proc
```

Figure 2: Generic evaluation function for RAP5; instantiated for "fill house".

where C and R stand for *Consumers* and *Resources* respectively. We also know that the following implications and equivalences hold for the basic RAPs (we write C_a and R_a for "assigned consumers" and "assigned resources" respectively):

$$\text{RAP2} \Leftrightarrow \text{SIZE}(C) \geq \text{SIZE}(R_a) \wedge \text{SIZE}(C) = \text{SIZE}(C_a)$$
$$\text{RAP3} \Rightarrow \text{SIZE}(C_a) \geq \text{SIZE}(R_a)$$
$$\text{RAP4} \Rightarrow \text{SIZE}(C_a) \leq \text{SIZE}(R_a)$$
$$\text{RAP5} \Leftrightarrow \text{SIZE}(C) \leq \text{SIZE}(R_a) \wedge \text{SIZE}(R) = \text{SIZE}(R_a)$$

It follows that

$$\text{RAP2} \wedge \text{RAP3} \wedge \text{RAP4} \wedge \text{SIZE}(C) = \text{SIZE}(R) \Rightarrow \text{RAP5}$$

Therefore, knowing that all rooms are inside the house and do not overlap, and the house is flat,

$$\text{SIZE}(Fp)^{point} = \text{SIZE}(Hs)^{point} \Rightarrow \text{"fill-house"}$$

```
More EFFICIENT evaluation function
for ''fill house'':

variables: out,r
constants: Fp

begin-proc
    out <- 0
    forall r in Fp
      out <- out + rect-length(r) * rect-width(r)
    end-forall
    return out
end-proc
```

Figure 3: More efficient evaluation function for "fill house".

That is, the objective of "fill house" coincides with the *number of house points being equal to the number of all room points in the floorplan.* Progress towards satisfying "fill house" can be measured by the total number of room points in the floorplan. MENDER searches its knowledge base of data types and finds that the number of points of an object of type rectangle can be efficiently computed as the product of rectangle length and rectangle width. By adopting this measure as the revised evaluation function for "fill house" (see Fig. 3) MENDER constructs an evaluation function that computes significantly faster the original one (Fig. 2).

Step 3 – Characterizing "improving operators". Next MENDER continues by working with the less efficient evaluation function in Fig. 2. Two additional pieces of information are associated with a generic evaluation function: the direction of change towards greater satisfaction of the constraint("increase" or "decrease"), and a characterization of events that can cause the desired change. For RAP5 the direction of positive change is "increase" (i.e. higher value of the evaluation function indicates greater constraint satisfaction), and the event to cause such change is *"increase the frequency of 'ASSIGN(c,r) = true' "*. For "fill house" this translates into the information that the value of the evaluation function can be increased (improved) if the floorplan is modified such that more frequently 'point-of-rectangle(p,rm) = true'. MENDER regresses this event through the definitions of relevant datatypes and predicates in its knowledge base, and thereby determines that only increases of the "length" and "width" parameters of the rooms in the floorplan can – in one application – lead to greater "filling" of the house. These parameter changes are termed "improving operators" to distinguish them from parameter value changes (e.g. changes in room location, "shrinking" of rooms) that are guaranteed *not* to contribute towards greater constraint satisfaction.

Step 4 – Instantiating the "patcher shell". The *patcher shell* is a piece of code that realizes a basic *hillclimbing* strategy (with backtracking). It is rendered operational for a given global constraint by instantiating it with the concrete evaluation function and "improving operator" information. The patcher shell provides the choice of two types of "greedy" control strategies: "greedy", and "greedy and strictly ascending". These controls differ in how and to what extent the evaluation function and "improving operator" information are employed. Both controls use the evaluation function to order the applicable patching operators at each choice point in decreasing order of progress towards satisfying the constraint. The "greedy and strictly ascending" control restricts the set of patching operators to only those that are "improving". For example, to patch for "fill house" possible parameter value changes are limited to increased values in the "length" and "width" parameters of rooms. In "greedy" patching, which operates with the full set of applicable operators (e.g. *all* changes of room "length" ,"width", and changes of the "x-coordinate" and "y-coordinate" of room location points) knowledge of "improving operators" can help to significantly reduce the cost of evaluating the promise of legal "next" operators at each decision point. Among the set of options only "improving" operators are evaluated in detail, while the closer examination of the remaining operators (a priori known to make no or negative progress) is suspended until after all prefered operators have failed to provide a solution. We found empirically that for up to 80% of all applicable operations the evaluation function value was never computed, reducing the cost/node considerably.

Step 5 – Building "generate-and-patch". MENDER interfaces a constrained generator with a patcher such that constraints satisfied by generation will not be violated during patching. This is accomplished by making the patcher adhere to the same restrictions that are forced upon generation when incorporating the local constraints. While these restrictions limit the set of patching operators, chances are that sufficient options remain to produce solutions. It is also exactly because of these restrictions that the patcher search space is typically smaller than the original generation space, allowing faster problem solving by patching than backtracking and regenerating.

Experimental results.

We present the results of our preliminary empirical studies of the performances of the generate-and-patch problem solvers that MENDER constructed for the "fill house" and "no overlap" constraints.

"Fill house". In the generate-and-patch problem solver for "fill house" the (RICK-compiled) constrained generator guarantees that all generator outputs are floorplans with nonoverlapping rectangular rooms no smaller than 5× 5 units, and are lo-

cated inside a given house area and adjacent to at least one house wall. We compare performances of constrained generate-and-test (g&t), generate-and-patch with "greedy" control of patching (g&p:greedy), generate-and-patch with "greedy and strictly ascending" (g&p:gr-asc) control. As added control condition, we also test generate-and-patch without any informed control strategy (g&p:default). Our performance measure is the "repair effort" expended by each problem solver after the first candidate has been generated.[1] "Repair effort" for generation is measured in number of nodes (alternative selections of parameter values) expanded through backtracking and regeneration. Repair by patching is measured in number of nodes (modifications of parameter values) expanded within the patcher space.

We ran each problem solver 20 times for floorplans with 1, 2, 3, and 4 rooms. The constrained generator produced floorplans in random order. The house dimensions were chosen relative to the number of rooms, such that the smallest legal floorplan would cover ~ 30% of the house area. Our results, averaged over the 20 runs, are plotted in Fig.4. Overall we find

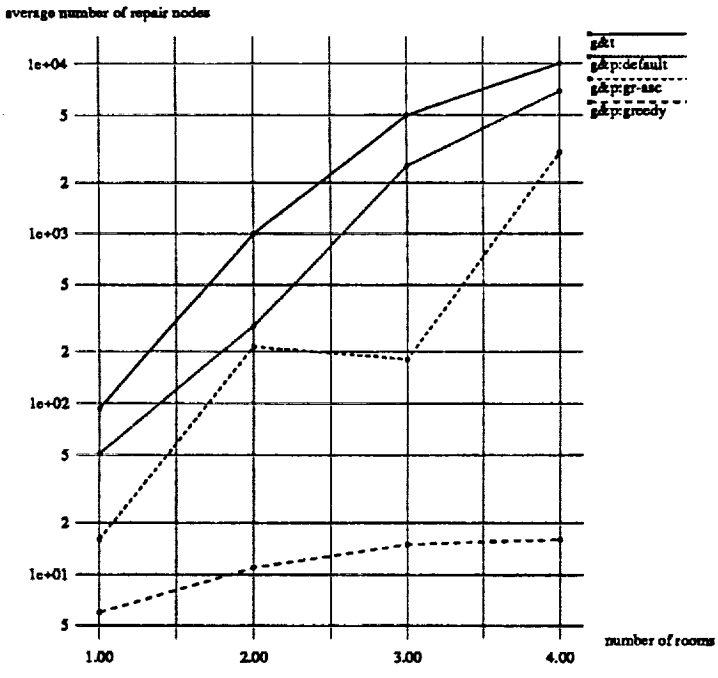


Figure 4: "Repair effort" of generate-and-test and generate-and-patch for "fill house" (y-axis scaled logarithmically).

that all types of generate-and-patch were consistently better than constrained generate-and-test which confirms our intuitions and observations. The plots show that generate-and-patch with "greedy" control very significantly outperformed constrained generate-and-test. E.g., for 4-room floorplan 10,000 generations[2] did not suffice to satisfy "fill house". In contrast, patching achieved repair in only 16 patcher nodes. Not surprising is the inefficient performance of generate-and-patch without control (g&p:def). Generate-and-patch with "greedy and strictly ascending" control was second best but notably less efficient than "greedy". In the past, we had shown that this type of problem solver can perform much better, when patching is interleaved with "block-preventing" moves that circumnavigate dead-ends [Voigt and Tong, 1989]. Naturally, this facility involves some additional cost which judging from the good performance of "greedy" was not warranted by our examples. At this point, we withhold judgment on the relative merits of "greedy" versus "greedy and strictly ascending patching". Which problem solver is likely to be more cost-effective seems to depend on the domain and the constraint. We need to study MENDER-compiled generate-and-patch problem solvers for larger numbers of more diverse constraints to obtain more conclusive results.

"No overlap". In a further study, we compare the performance of the MENDER-compiled greedy generate-and-patch problem solver (g&p:greedy) to satisfy "no overlap" with the RICK-compiled constrained generator (constr-g) which outputs nonoverlapping floorplans. We measure and compare the performances of both types of problem solvers in number of nodes expanded in the respective generation and patching spaces. We ran generate-and-patch and constrained generation 20 times for 2 to 5 rooms and a 15×15 house. Note that constraint incorporation by RICK does not prescribe a particular (constraint-dependent) generation order. Therefore, performance data were collected with a randomized generation order in the constrained generator. The results, averaged over 20 runs, are plotted in Fig. 5. We find that generate-and-patch finds solutions considerably faster than the constrained generator. These data show that the utility of generate-and-patch algorithms is not restricted to constraints whose extremely high degree of globality renders the compilation of a constrained generator infeasible or undesirable. Even when it is possible to compile a constrained generator, a generate-and-patch algorithm may be a viable, and potentially preferable alternative.

Applying MENDER to other domains.

At the present time all problem solvers that MENDER has automatically constructed solve problems in the domain of 2D-floorplanning. However, we have worked

[1] Generating the first candidate has a fixed cost shared by all problem solvers, and is therefore ignored in our study of comparative cost.

[2] Although the plot shows data point 10,000, patching was cut off at 10,000 nodes without solutions.

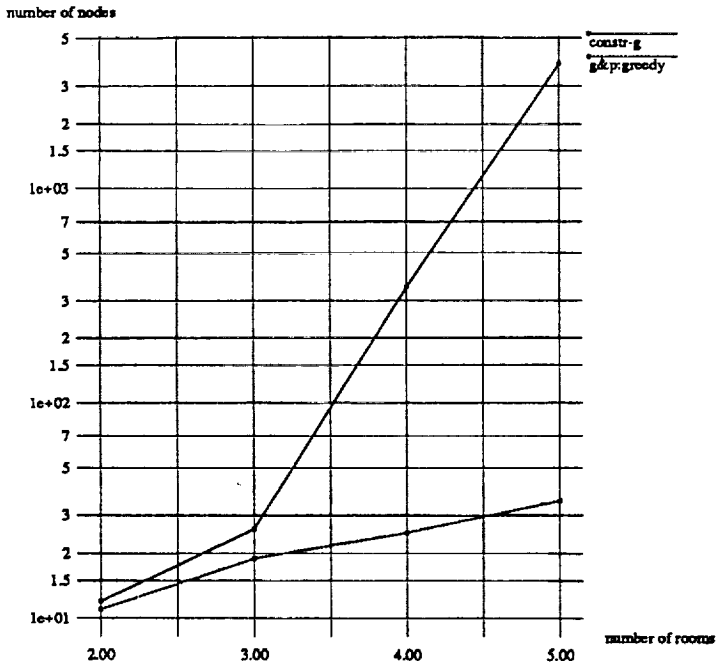


Figure 5: Satisfying "no overlap" by constrained generation vs. generate-and-greedy-patch (15x15 house) (y-axis scaled logarithmically).

out paper traces on how an extended version of our system will be able to compile generate-and-patch problem solvers for constraints in a variety of other domains (e.g. n-queens, graph-coloring, scheduling, VLSI-design) [Voigt, 1991]. Here we will briefly summarize how MENDER could be applied to a *multiprocessor scheduling* problem taken from the listing of NP-complete problems in [Garey and Johnston, 1979]:

> Multiprocessor Scheduling:
>
> "INSTANCE: Set T of tasks, number $m \in Z^+$ of processors, length $l(t) \in Z^+$ for each $t \in T$, and a deadline $D \in Z^+$.
>
> QUESTION: Is there a m-processor schedule for T that meets the overall deadline D, i.e., a function $\sigma : T \rightarrow Z_0^+$ such that, for all $u \geq 0$, the number of tasks $t \in T$ for which $\sigma(t) \leq u < \sigma(t) + l(t)$ is no more than m and such that, for all $t \in T, \sigma(t) + l(t) \leq D$?"

Suppose that this problem is presented to MENDER in terms of a consumer structure *Schedule* which is a set of *tasks* with known lengths and a resource structure *TimeTable* which is a list of consecutive *time slots*. The latest time slot corresponds to the deadline D. Assume that a generator produces schedules by assigning starting times to each task. To satisfy scheduling constraint a schedule must satisfy the following conjunction of subconstraints: (1) a task in *Schedule* must be assigned a starting time that corresponds to a time slot in *TimeTable*, and (2) a time slot in *TimeTable* must not be assigned to more than m tasks. (Note that we model the m processors as a *capacity limitation* on each time slot.)

MENDER would recognize this constraint as an instance of RAP7 which is a conjunction of RAP2 and RAP4$^+$ where RAP4$^+$ is a generalization of the original RAP4. RAP4 requires that each resource is assigned to at most 1 consumer. RAP4$^+$ requires that each resource be assigned to at most m consumers for $m > 2$. Based on this classification of the constraint, MENDER would derive as an evaluation function a measure that *combines* the number of all tasks in *Schedule* assigned to time slots *within* TimeTable with the sum of how many times exceeding m each time slot has been assigned to some task. Given this evaluation function, MENDER determines that assigning *earlier starting times* is most likely to improve a schedule with respect to the evaluation function. Therefore, generate-and-patch with *greedy* patching would prefer scheduling tasks earlier over rescheduling tasks to later starting times.

Related research.

KIDS [Smith, 1991] and STRATA [Lowry, 1991] are two algorithm design systems that are closely related to MENDER. Both systems design search algorithms but differ from MENDER in the assumptions made about the initial problem specification, and in the way domain knowledge and algorithm knowledge are used to construct search operators and control facilities.

KIDS automatically constructs search algorithms, e.g. a *global search algorithm*, by retrieving from a library of abstract global search theories a theory that applies to the datatypes mentioned by the constraint. The abstract theory is then specialized into a global search algorithm through a series of program-transformations. Selection of global search theories and transformation steps is done in interaction with the user. KIDS enables the search algorithm to make use of problem-specific information by deriving *necessary filters* that prune those parts of the search space that are void of solutions. The derivation of necessary filters is accomplished by a *deductive inference* component.

KIDS is a much larger and more general algorithm design system than MENDER. KIDS works with a larger and more varied library of abstract search theories, enabling it to not only construct global search algorithms, but local search and divide-and-conquer problem solvers as well. MENDER is restricted to compiling local search algorithms, and does so only for constraints that fall into one of 16 abstract RAP categories. However, precisely because of its restrictions, MENDER has several advantages over KIDS. MENDER is fully automatic whereas KIDS requires

that the user make important design decisions. Because MENDER's constraint knowledge is restricted to RAPs for which generic evaluation functions are known, the cost of compiling search control facilities, i.e. the cost of retrieving and instantiating an evaluation function schema, is relatively cheap in MENDER. The derivation of necessary filters by KIDS's deductive inference component can be very costly. For constraints that are as global as "fill house", we expect KIDS to have great difficulty in deriving a filter. MENDER, however, can easily and cheaply provide an evaluation function to guide the search.

The STRATA system by Lowry has been integrated into KIDS as the component which derives *local search* problem solvers. STRATA and MENDER are similar in that they derive search operators ("patching operators" in MENDER; "neighbourhood structures" in STRATA) from datatypes mentioned in the constraint formulation and a *cost function* whose value local search strives to optimize. A major difference between both systems lies in the nature of the initial problem formulation. STRATA accepts optimization problems that list output conditions and a cost function as two separate and independent components of the problem formulation. In principle, any set of output conditions could be paired with any cost function. MENDER's style of problem formulation offers less flexibility, in that the the output conditions and cost function ("evaluation function") are interdependent. In MENDER, a constraint is presented only in the form of output conditions. A suitable cost function is then *derived* from the output conditions. As in comparison with KIDS, MENDER trades flexibility and variety of problem classes for a more direct and *low cost* algorithm design process. Since MENDER's cost functions are instances of generic cost functions, the type of cost function is known to the system. Knowing the nature of the cost function a priori allows us to equip MENDER with a regression mechanism that is specialized – and therefore cost-effective– in tracing desirable cost function changes back to local modifications in the solution structure. For the similar task, STRATA needs to use the much more general and costly deductive inference component of KIDS.

MENDER-compiled patchers adopt a repair strategy that is similar to the one recently examined by Minton [Minton *et al.*, 1990]. Minton demonstrates how a local search problem solver controlled by a simple "minimize conflicts" heuristic can solve large-scale scheduling and very large n-queens problems in approximately linear time with respect to problem size. Socic and Gu [Sosic and Gu, 1990] reported comparable performances for similar local search problem solvers for very large n-queens problems. However, to automate the construction of efficient problem solvers that can take advantage of the "minimize conflicts" heuristic, a constraint has to lend itself to an easy quantification in terms of number of "conflicts". The notion of "conflict" associated with a given constraint may or may not be obvious from the formulation of the constraint. MENDER solves both these problems for constraints of type RAP. MENDER reexpresses RAP constraints in terms that allow the conceptualization of a notion of "conflict" that captures the specifics of the constraint and is amenable to easy quantification.

Future research.

In the near future, we plan to extend MENDER to handle global constraints in a variety of other domains, e.g. scheduling, VLSI-design, n-queens, graph-coloring, satisfiability. We also intend to explore possibilities of applying MENDER's classification-based approach to automatically compiling "look-ahead" facilities which detect and circumnavigate unpatchable states early on.

Acknowledgements.

I am grateful to Chris Tong for his insights and guidance. For valuable comments and suggestions I also thank Lou Steinberg, Don Smith and Tom Ellman. Further thanks to Wes Braudaway for the constrained generators constructed with his RICK compiler.

References

Braudaway, W.K. 1991. *Knowledge Compilation for Incorporating Constraints.* Ph.D. Dissertation, Rutgers University.

Garey, M.R. and Johnston, D.S. 1979. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Freeman.

Lowry, M.R. 1991. Automating the Design of Local Search Algorithms. In Lowry, M.R. and McCartney, R.D., editors 1991, *Automating Software Design.* Menlo Park: AAAI Press.

Minton, S.; Johnston, M.D.; Philips, A.B.; and Laird, P. 1990. Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method. In *Proceedings of AAAI-90.*

Smith, D.R. 1991. KIDS – A Knowledge-Based Software Development System. In Lowry, M.R. and McCartney, R.D., editors 1991, *Automating Software Design.* Menlo Park: AAAI Press.

Sosic, R. and Gu, J. 1990. A Polynomial-Time Algorithm for the N-Queens Problem. *SIGART Bulletin* 1(3).

Tong, C. 1991. A Divide-and-Conquer Approach to Knowledge Compilation. In Lowry, M.R. and McCartney, R.D., editors 1991, *Automating Software Design.* Menlo Park: AAAI Press.

Voigt, K. and Tong, C. 1989. Automating the Construction of Patchers that Satisfy Global Constraints. In *Proceedings of IJCAI-89*, Detroit.

Voigt, K. 1991. Working Notes. Computer Science Department, Rutgers University.

CARDS: A Blueprint and Environment for Domain-Specific Software Reuse

Kurt C. Wallnau, Anne Costa Solderitsch and Catherine Smotherman
Paramax Systems Corporation
(A Unisys Company)
Farimont, West Virginia and Paoli, Pennsylvania

CARDS (Central Archive for Reusable Defense Software) exploits advances in domain analysis and domain modeling to identify, specify, develop, archive, retrieve, understand and reuse domain-specific software components. An important element of CARDS is to provide visibility into the domain model artifacts produced by, and services provided by, commercial computer-aided software engineering (CASE) technology. The use of commercial CASE technology is important to provide rich, robust support for the varied roles involved in a reuse process. We refer to this kind of use of knowledge representation systems as supporting "knowledge-based integration."

1. Introduction

The problem of achieving satisfactory levels of reuse in the development of defense software has been challenged in recent years, but with limited success. A development which will surprise no one in the AI community is a recent focus by the US DoD on attacking the reuse problem on a per-domain basis. A notable example is the CAMP project [1]. CARDS (Central Archive for Reusable Defense Software) attempts to exploit advances in *domain analysis* and domain *knowledge representation* to identify, specify, develop, archive, retrieve, understand and reuse domain-specific software components* — and to do so in a way that is independent of the underlying application domain.

We view the domain analysis and domain knowledge representation as the key to achieving the CARDS objectives — with special emphasis on *understanding* the relationships between software components and the domain model. However, the stipulation that CARDS should be applicable across a variety of application domains has interesting consequences on the construction of a blueprint and environment for domain-specific reuse.

2. Divergent Roles and Environments

The defense department develops systems spanning many domains — exactly how many is a matter of contention and will only be resolved when a concise definition of domain is available and is applied to defense department procurements. Software continues to be a critical component of systems developed in most of these domains. Attaining high-leverage reuse within narrowly focused application domains is well-justified by research, experience and economics. However, to institutionalize domain-specific reuse, a blueprint detailing how to undertake the development of a domain-specific reuse library, and a computer-aided support environment for putting the blueprint in action, is necessary.

The problem confronted by CARDS is the multiplicity and divergence of dimensions, or elements, of any CARDS architecture†. For example, CARDS must support a variety of roles, where roles are task-related personifications of activities necessary to achieve reuse. Examples include: performing domain analysis; using the results of a domain analysis (i.e., the domain model) to identify abstract interfaces; specifying the concrete interfaces; implementing the components; designing a user-friendly library classification scheme; archiving components within the classification scheme; and, ultimately, the end-user role of locating and retrieving components. Figure 2-1 illustrates a strawman architecture for a CARDS environment.

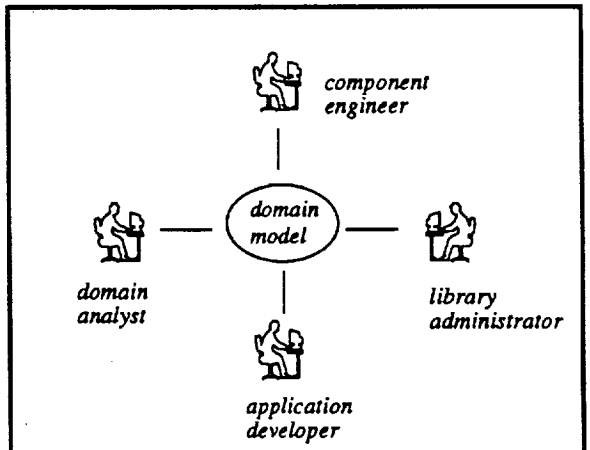


Figure 2-1 Central, Shared Domain Model

Of course, Figure 2-1 is overly simplistic. Since each of these roles represents a different perspective (and several roles are missing), different processes, methods and support technology will need to be brought to bear to support

*. Software components include assets such as requirements and design models, parts generators, programs, etc. See [2] for more details.

†. We use the term "architecture" to refer to the *blueprint* and support technology.

different kinds of tasks. For example, the kinds of information produced and consumed by a domain analyst will be different from that produced and consumed by a component engineer. One way to address divergent roles is to provide alternative views into a shared knowledge base, as illustrated in Figure 2-2. This is the approach that is taken

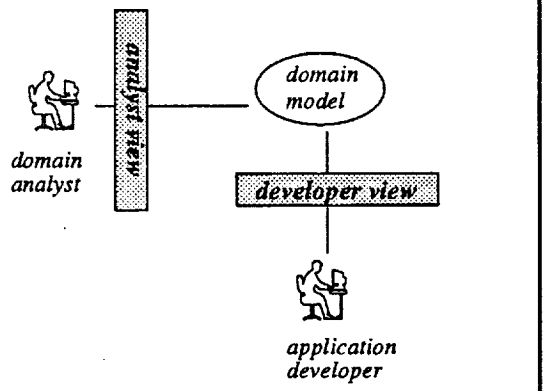


Figure 2-2 Uniform Domain Model & Views

in classical software development environment architectures [3] as well as hypertext-oriented knowledge representation frameworks [16].

Of course Figure 2-2 is also overly simplistic. First, there is no consensus regarding domain analysis process, method or representation. It appears that the choice of domain analysis technique depends to some extent upon the desired end-result of the analysis — e.g., supporting reuse, understanding a system, comparing different systems, etc. For example, Diaz's analysis technique [4] for reuse differs substantially from Brown's informal [5] technique for comparing software environment architectures, while LaSSIE makes use of a uniform, formal knowledge representation scheme for managing the complexity of a layered system [6].

Second, domain analysis techniques will vary across application domains. For example, information management application domains may be suitably modeled using classical structured analysis and structured design techniques; real-time systems may require the addition of behavioral models and temporal logics; complex, interactive systems may be best modeled using object-oriented techniques. While, in theory, each of these techniques has an analogue in a more generic knowledge representation formalism, such a mapping would not be practical.

Third, even within isolated application domains it may be useful to employ a variety of domain analysis and representation techniques. For example, the SEI feature-oriented domain analysis method (FODA) [7] employs an eclectic assortment of representations. Besides FODA, the notion of refinement, crucial in various formal design methods, implies a mapping among various representations, for example Z [17] specifications to program source. Thus, focusing on support for the domain analyst role, a more realistic CARDS architecture is illustrated in Figure 2-3.

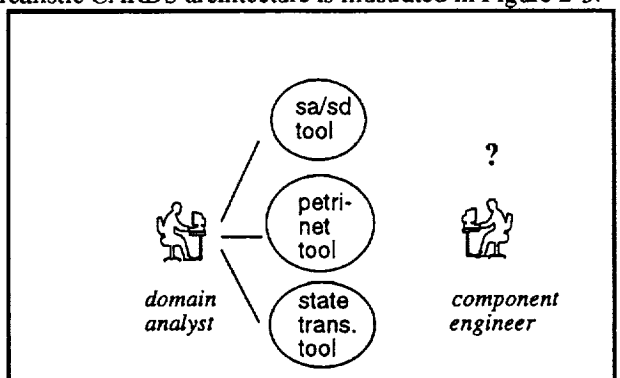


Figure 2-3 Eclecticism and Tool Coalitions

There is an underlying pragmatic basis for Figure 2-3 as well — while domain analysis and knowledge representation are better understood today than just a few years ago, the technology is still unstable. Further, there is an existing body of commercial CASE tools available which can support practical application of domain analysis techniques.

There are severe problems underlying Figure 2-3. The collection of analysis tools employed by the domain analyst — in essence the domain analysis environment — are not likely to be well integrated with respect to the domain analysis process, the logical services provided by the tools, nor the underlying tool mechanisms [8]. The tools themselves are at worst completely egocentric and at best wired together in some loose form of tool coalition [9]. This makes it difficult to verify the completeness and consistency of domain models.

Just as serious is the lack of integration of the domain analysis environment with the environment required by the component engineer. Not only will it be difficult for the component engineer to locate and understand the portions of the domain model relevant to the construction of software components, but the component engineer will also have specialized tools to support development tasks, e.g., coding, performance, annotation, testing and configuration management tools. The conceptual distance between the analysis tools and development tools makes even tool coalitions an unlikely prospect. A similar impedance mismatch exists between other roles in the CARDS architecture.

3. Knowledge-Based Integration

Figures 2-2 and 2-3 illustrated the dichotomy between an idealized view of a domain-specific reuse environment, and the view most likely to emerge from the combination

of state-of-the-practice tool support and the requirement for domain-independence of the CARDS architecture. These views need to be merged. That is, we must provide a semantically meaningful view, for each role, into a domain model, while not sacrificing the tool support necessary to support the processes associated with a particular role.

Our approach is to merge these views, initially using the STARS* Reusability Library Framework (RLF) [10] as a meta-model for relating, and integrating, services provided by and artifacts produced by different tools. The hybrid knowledge-representation system in RLF combines a semantic network system based upon KL-ONE, with an extensible, *typed* rule-base system. A high-level view of this architecture is depicted in Figure 3-1.

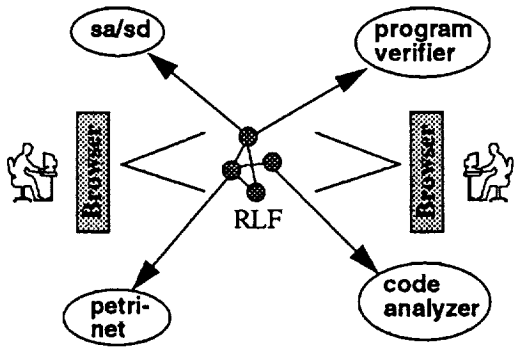


Figure 3-1 Meta-Model Integration in RLF

The architecture highlighted in Figure 3-1 has several interesting properties. First, the RLF knowledge base provides a single meta-model which a) uses the semantic network to relate the artifacts produced by various tools, and b) uses *action* rule types to tie tool services to tool artifacts†. The first property increases the visibility to relationships among elements of the domain model that are created by one role but semantically meaningful to other roles in the CARDS environment. The second property leverages the substantial investment in existing CASE technology and preserves a convenient, comfortable and functional environment already tailored to role-specific processes.

Second, the browser allows various users in the CARDS environment to view only those portions of the knowledge base that are appropriate for their role. Two forms of view filters are possible: through the graphical browser, and through the use of *advisor* librarians (also available through the browser). The former is a relatively straightforward user-interface problem. The latter is supported through the use of various rule types which are used by a special advisor inference engine — TAU.

Third, the use of RLF provides the basis for the development of other specialized types of inferencers to support the reuse process. One inferencer — Gadfly [11], has already been prototyped to support component specification and qualification. Other inferencers have been developed using a similar hybrid knowledge representation system for systems diagnostic maintenance [12] and (more closely related to software component reuse) hardware configuration [13].

4. CARDS and the Reuse Process

The architecture in Figure 3-1 is sketchy and only briefly discussed because the real problem is not the mechanisms of the CARDS environment, but the use of it within the context of an overall reuse process. A number of questions will need to be answered, perhaps some of them on a per-domain basis:

- How much of the domain model should be captured in the knowledge base, versus its use as an index into tool artifacts and tool services?
- What are the appropriate views into the knowledge base? For example, should an application developer's view be based upon models of architectures [14] or requirements [15]?
- When is a domain *ripe* for reuse [2]?

While these discussions have focused on the integration of different user roles with a reuse repository‡, another dimension of integration can be found when viewing a domain-specific reuse library as a bridge between supply-side and demand-side reuse processes. As illustrated in Figure 4-1, the scope of a repository can vary according to the nature of the domain analysis processes, e.g., how close is the "fit" between the domain analysis process and the domain modeling services provided by the repository, and the nature of the demand-side processes, e.g., who on the demand side will be using the repository?

In Figure 4-1, two parallel life-cycle processes are depicted: domain engineering and software engineering represent the supply-side and demand-side reuse processes, respectively. The repository can be scoped to capture the by-products of different domain engineering subprocesses; such decisions about scoping can result from, or can result

*. STARS — Software Technology for Adaptable, Reliable Systems.

†. A similar integration approach is provided in Frame Technology's *Live Links* and in several other systems.

‡. The use of a domain model as a kind of repository has been implicit throughout the discussion. The terms "archite," "library" and "repository" are also used synonymously.

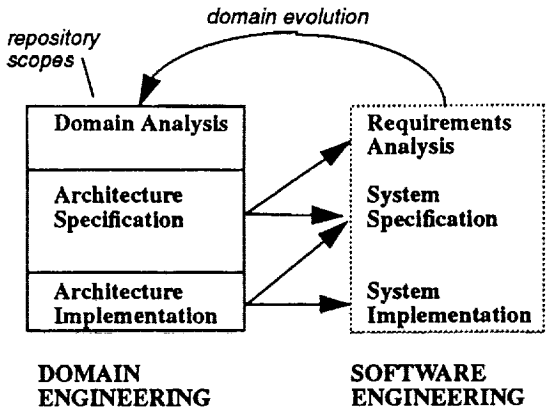


Figure 4-1 Repository Scopes and Process

in, different demand-side processes. For example, scoping the repository to include only the implementation components produced by domain engineering processes will result in a "conventional" parts library. Such a design decision can be motivated by various factors, including the possibility that the demand-side processes are still too chaotic to support more systematic reuse. Thus, in Figure 4-1 the "parts" library could support, at worst, ad hoc opportunistic reuse during system implementation, and, at best, could support a system specification that takes some advantage of existing reuseable components.

There are clearly potential advantages to extending the scope of the repository to address the entire spectrum of domain-engineering by-products, including domain analysis. In Figure 4-1 the primary benefit illustrated is the potential for closing the loop between domain engineering and software engineering through a feedback and domain-evolution path. Such a feedback loop can probably only be supported if the domain model is captured and represented in a reasonably formal manner.

5. Summary

We have described the problem of constructing an environment to support the construction of domain-specific reuse libraries in terms of integration. The integration problem involves integration of:

- roles in the reuse process
- domain analysis tools with each other
- domain analysis tools with a reuse process

We briefly outlined the use of a hybrid knowledge representation system, RLF, to act as an integrating agent to provide role-specific views into the domain model, and to support the use of an eclectic assortment of modeling techniques by tapping into a large, robust CASE market.

The CARDS program will focus, in the next year, on creating a *blueprint* for achieving reuse in the DoD. This blueprint will address technical as well as non-technical issues, and will provide guidelines for the use of a hybrid knowledge-representation/CASE tool architecture for developing domain-specific reuse libraries and using domain-specific software architectures and assets to create application systems.

The CARDS program will also be experimenting with domain-specific reuse environment and system composition techniques tailored to the command center subdomain of C^2 applications. The conceptual model for this composition is similar to that of hardware configuration [13] — a user *configures* a system of software components based upon a inferencer-directed dialogue designed to elicit system requirements.

References

[1] Anderson, C.M., McNicholl, D.G., "Common Ada Missile Packages (CAMP); Preliminary Technical Report, Vol. 1," in *STARS Workshop Proceedings*, April 1985, FO 8635-84-C-0280.

[2] Simos, M.A., "The Growing of an Organon: A Hybrid Knowledge-Based Technology and Methodology for Software Reuse," *Domain Analysis and Software Systems Modeling*, Prieto-Diaz, Arango, Eds., IEEE Computer Society Press, ISBN 0-8186-8996-X.

[3] *Integrated Project Support Environments: The Aspect Project*, A.W. Brown (Editor), Academic Press, 1990.

[4] Prieto-Diaz, R., "Domain Analysis for Reusability," *Domain Analysis and Software Systems Modeling*, Prieto-Diaz, Arango, Eds., IEEE Computer Society Press, ISBN 0-8186-8996-X.

[5] Brown, A.W., Feiler, P.H., *An Analysis Technique for Examining Integration in a Project Support Environment*, Technical Report CMU/SEI-92-TR-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, January 1992.

[6] Devanbu, P., Brachman, R.J., Selfridge, P., Ballard, B., "LaSSIE: A Knowledge-Based Software Information System," *Domain Analysis and Software Systems Modeling*, Prieto-Diaz, Arango, Eds., IEEE Computer Society Press, ISBN 0-8186-8996-X.

[7] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.

[8] Brown, A.W., Feiler, P.H., Wallnau, K.C., "Understanding Integration in a Software Development Environment," accepted for *Second IEEE International Conference on Systems Integration*, Irvine, CA, for May, 1992.

[9] Brown, A.W., Feiler, P.H., Wallnau, K.C., "Past and Future Models of CASE Integration," submitted to *Fifth International Workshop on Computer-Aided Software Engineering*, for July 1992.

[10] Solderitsch, J., Wallnau, K., Thalhamer, J., "Constructing Domain-Specific Ada Reuse Libraries," in *Proceedings of the 7th Annual National Conference on Ada Technology*, Atlantic City, NJ, 1989.

[11] Wallnau, K., Solderitsch, J., Thalhamer, J., et. al., "Construction of Knowledge-Based Components and Applications in Ada," *Special Issue of the IEEE Intelligent Systems Review*, Spring 1989.

[12] Matuszek, P., Clark, J., Sable, J., Corpron, D., Searls, D., "KSTAMP: A Knowledge-Based System for the Maintenance of Postal Equipment," United States Postal Service Advanced Technology Conference, May 1988.

[13] Searls, D., Norton, L., "Logic-Based Configuration with a Semantic Network," in *The Journal of Logic Programming*, Volume 8, 1990, pages 53-73.

[14] D'Ippolito, "The Context of Model-Based Software Engineering," in *Proceedings of the Workshop on Domain-Specific Software Architectures*, Hidden Valley, PA, DSSA Program Manager, DARPA/ISTO, 1400 Wilson Blvd., Arlington, VA 22209, July 1990.

[15] *A Domain Analysis Process, Interim Report*, Domain_Analysis-90001-N, Version 01.00.03, Software Productivity Consortium, 2214 Rock Hill Road, Herndon, VA, 22070, January 1990.

[16] Mayfield, J., Nicholas, C., *Using Semantic Networks to Enrich Hypertext Links*, Technical Report, Computer Science Dept., University of Maryland Baltimore County, January, 1992.

[17] *Z — An Introduction to Formal Methods*, Diller, A., John Wiley and Sons, ISBN 0-471-92489-X.